A UML-based static verification framework for security

Requirements Engineering manuscript No.
(will be inserted by the editor)

Igor Siveroni · Andrea Zisman · George Spanoudakis

A UML-based Static Verification Framework for Security

the date of receipt and acceptance should be inserted later

Abstract Secure software engineering is a new research area that has been proposed to address security issues during the development of software systems. This new area of research advocates that security characteristics should be considered from the early stages of the software development life cycle and should not be added as another layer in the system on an ad-hoc basis after the system is built. In this paper we describe a UML-based Static Verification Framework (USVF) to support the design and verification of secure software systems in early stages of the software development life-cycle taking into consideration security and general requirements of the software system. USVF performs static verification on UML models consisting of UML class and state machine diagrams extended by an action language. We present an operational semantics of UML models, define a property specification language designed to reason about temporal and general properties of UML state machines using the semantic domains of the former, and implement the model checking process by translating models and properties into Promela, the input language of the SPIN model checker. We show that the methodology can be applied to the verification of security properties by representing the main aspects of security, namely *availability*, *integrity* and *confidentiality*, in the USVF property specification language.

Keywords UML · Model Checking · SPIN

1 Introduction

Secure software engineering is a new research area that has been proposed to address security issues during the development of software systems [25]. This new area of research advocates that security characteristics should be considered from the early stages of the software development life cycle and should not be added as another layer in the system on an ad-hoc basis after the system is built. More specifically, security software engineering attempts to fulfill the lack (a) in existing approaches, techniques, and methodologies in the area of software engineering to provide support for the analysis and design of security requirements and properties, and (b) in existing approaches for security engineering which concentrate on security issues and consider limited aspects of the software system as a whole.

In the last few years, various approaches to support formal verification techniques for security protocols [1, 10, 23] have been proposed. However, existing formal verification techniques for security are (i) limited, as they focus on the verification of mainly interaction protocol designs, (ii) cannot guarantee security properties of protocol implementations, (iii) do not consider the system as a whole, and (iv) use disjoint security and system design models that are typically expressed in different languages [34]. As suggested in [2, 5, 26], security should be considered from the early stages and through all the stages of software development. Therefore, it is necessary to develop verification approaches supporting the specification and analysis of security aspects during early stages of the development life-cycle, and in a way that takes into account the entire system design rather than as a separate layer added to the system as an afterthought in the form of security protocols.

In this paper we describe a UML-based Static Verification Framework (USVF) to support the design and verification of secure software systems in early stages of the software development life-cycle taking into consideration security and general requirements of the software system. The development of USVF has been driven by requirements and scenarios identified by industrial partners in the areas of media and security in an European project called PEPERS focusing on mobile security [32]. USVF uses UML models represented as class and state machine diagrams, and a specific action language based on *guards* and *effects* to allow designers of the software system to express extra behaviour specifications. The framework incorporates a property language that allows

Department of Computing, City University London
Northampton Square, London EC1V, UK
E-mail: {siveroni,A.Zisman,G.Spanoudakis}@soi.city.ac.uk

Table 1 System Requirements

	Requirements
R1	Manager starts process by assigning roles and task to logged-in peers
R2	A Peer can adopt a Journalist or Photographer role
R3	Functionality should be provided to allow Peers to exchange messages between each other
R4	All Peers should report completion of task to the Manager
R5	A Peer cannot change roles
R6	Roles and tasks should always (and only) be assigned by the Manager
R7	Peers should not exchange information, unless authorised, during the execution of the assigned task

the user to specify properties that need to be satisfied by the system in linear temporal logic. The main novelty of the property language of USVF arises from the enhancement of the basic underlying linear temporal logic with object oriented modelling constructs, namely attributes, events and actions. The properties that can be expressed in this language refer to events and actions affecting the state of system objects and the reasoning underpinning the checks for the satisfaction of the properties takes into account behavioural system models expressed as state machines. This reasoning is based on model checking that is performed using SPIN after translating the properties and the model of a system into PROMELA (*i.e.* , the specification language of SPIN). Furthermore, USVF translates the results of simulation runs and model checks performed by SPIN into execution traces of the UML state machines to make them legible for system developers.

The main contribution of this paper is the definition of the property specification language that deals directly with UML models and can be used to express - explicitly in UML - properties about the execution of UML models and state machines. The property language of USVF enables the specification of the basic security properties of confidentiality, integrity and availability. More generally, however, it can express properties concerned with the order of execution of transitions, invocation of actions, and their effects onto the state of the system. Considerable amount of research (section 6) has been dedicated to the development of formal operational semantics of UML and the verification of UML models using model checking techniques. However, less effort has been dedicated to the development of property specification methodologies suited to model checking of UML (both syntax and semantics). An initial verification property language, μ-UCTL, that considers the semantics of UML has been proposed in [9]. USVF, however, provides a property specification language with a richer object oriented syntax and corresponding operational semantics enabling the expression of a richer set of predicates involving object fields, class fields and action events.

The work presented in this paper has two goals: (1) to bridge the gap between UML semantics and property specification of UML models, and (2) to show that the methodology can be applied to the verification of security properties. In order to achieve the first goal, we define a property specification language designed to reason about temporal and general properties of UML state machines using the semantic domains of the former, and implement the model checking process by translating models and properties into Promela, the input language of the SPIN model checker. Finally, we show that the developed framework can be applied to the verification of security properties by demonstrating how the basic security properties, namely *availability*, *integrity* and *confidentiality*, can be expressed in the USVF property specification language.

1.1 Static Verification Framework (USVF) Overview

The general architecture of USVF is shown in Figure 1. As shown in the figure, USVF consists the following components: Design Model Constructor, Property Editor, Verifiers, Translator, and Results Visualization. The components and the interactions between them can be seen in Figure 1.

The *Design Model Constructor* component is responsible for the construction of abstract design models of the system. We use UML models [28] for the specification of the structural and behavioural elements of the systems to be verified. We integrate an existing UML case tool to assist with the construction of such design models.

The *Property Editor* allows the user to build the properties to the verified by the USVF. These properties are specified using an extended and user-friendly version of linear temporal logic (LTL) [6] tailored to reason about objects and state machines.

We have chosen to use model checking as our main verification technique and, in particular, the *Verifiers*

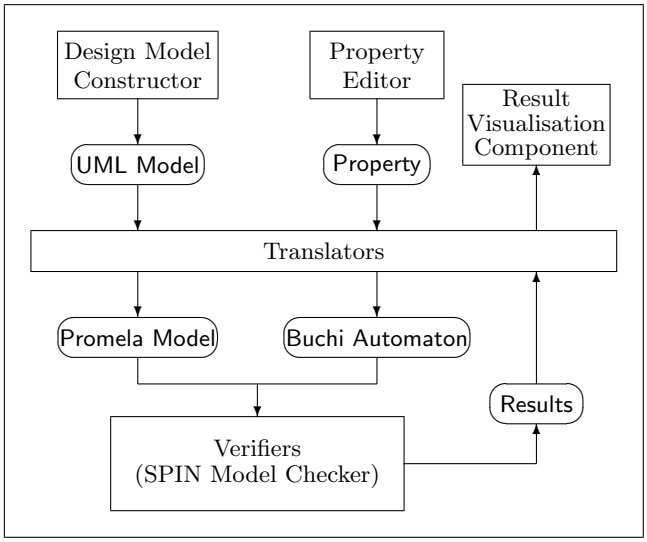

Fig. 1 UML Static Verification Framework

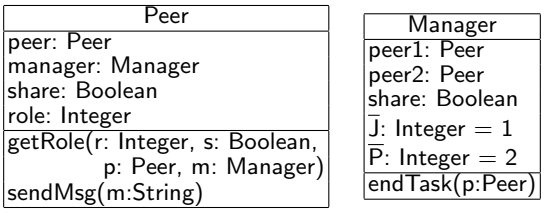

Fig. 2 Example: UML Class Diagram

component uses SPIN [12], a generic model checking tool that has been applied to the verification of several control and software systems. SPIN's specification language, Promela, is similar to C and supports message passing channels, essential for the modeling of distributed systems. Furthermore, SPIN uses an on-the-fly model checker thus avoiding the need to construct a global state graph and, consequently, the state-explosion problem. The mix of flexibility and efficiency, together with its ability to specify properties as LTL formulas and automata, makes SPIN an ideal target system.

In order to present the results of the verification process, the framework contains a *Result Visualization Component* that shows the parts of the design models involved in a property violation. This is achieved by displaying a step-by-step execution trace of simulations and error trails.

Finally, the framework contains *Translators* to support the mappings from UML design models into Promela models, and from properties expressed in the extended LTL property language into the specification language used by the verifiers.

1.2 Motivating Example

In order to illustrate the key concepts of our approach, let's consider an example of a peer-to-peer system designed to support the exchange tasks and data between journalists and photographers working in a media company. This example has been extracted (and adapted) from scenarios specified by the industrial partners of the PEPERS project [32].

Our example is composed of two peers running on mobile devices of journalists and a peer system used by their manager. The journalists and the manager in the scenario participate in the news coverage of an event. Once an event has been identified, the manager starts the news coverage process by assigning specific event coverage roles (e.g. reporting journalists, photographer) to each of the peers currently logged-in and sending them information related to the event. Peers can be assigned one of two roles, journalist or photographer.

After logging into a peer group, a peer should wait for the manager to send the event coverage role and information related to the task to be performed. After this information has been sent by the manager the peer can proceed with actions related to event coverage. Table 1 shows a list of requirements for this system, including requirements related to security.

For example, role assignment should always be performed by the manager and peers cannot modify their roles once they have been assigned. Furthermore, in order to ensure independent coverage of the event, the journalist and photographer covering the event should not exchange information about the event during the execution of the assigned task unless previous authorisation has been granted.

Security properties address three very important aspects of system behaviour[33], namely, confidentiality, integrity and availability:

- Confidentiality, also known as secrecy or privacy, ensures that data is only made available to authorised parties (R6, R7).
- Integrity ensures that data can only be modified by authorised parties or only in authorised ways (R5).
- Availability ensures that data and functionality are available to authorised parties at appropriate times. It's opposite is sometimes called denial of service (R4).

In the rest of this paper, we will show how USVF can be used to support the specification and checking of the satisfiability of the above types of security properties and in reference to a design model of the above system that is expressed in UML. Figure 2 shows classes in the design of this system that represent peers and Figure 3 the state machines defining the behaviour of these peers.

Before doing this, however, we present the general architecture of USVF to enable an understanding of the overall context of its functionalities.

1.3 Outline

The rest of this paper is structured as follows. Section 2 defines the structure of UML models considered by USVF and formalises operational semantics of state machines. Section 3 presents the syntax of the property specification language in USVF and the semantics of UML model checking. Section 4 describes how to translate UML models and properties into Promela models and SPIN LTL, respectively. Section 5 presents implementation aspects. Section 6 compares the framework with related work in the field. Finally, Section 7 discusses the contributions and limitations of the framework, and possibilities for future work.

2 UML Models and Semantics

2.1 Model Definition

The structure of the UML models considered in this paper is defined in Figure 4. These models are graphically

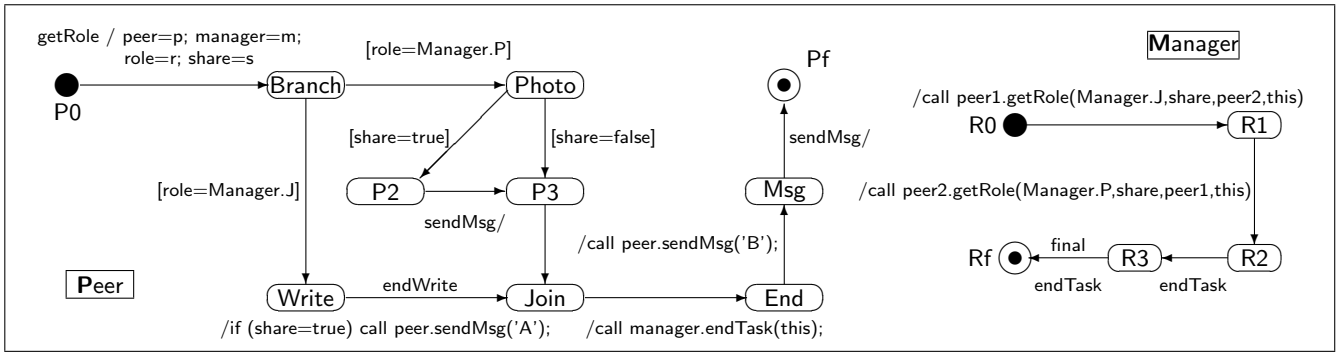

Fig. 3 Peer/Manager State Machines

represented by two types of UML diagrams: class diagrams, which define the structure of the model, and state chart diagrams, which specify the behaviour of each of the defined classes. A valid UML model, made of a single class diagram and a single state chart diagram per class, must be a correctly typed element of Model.

The UML subset used by the USVF is expressive enough to allow the user to model standard object-oriented elements such as classes, objects with static and non-static fields (attributes), as well as complex object behaviour (including state updates, iterations, conditional transitions, and hierarchical state machine diagrams.) The subset of UML assumed by USVF allows also the specification of interactions between state machines via message-passing.

Formally, a model U is made of a set of class and object declarations. Class declarations (c) correspond to the classes defined by the class diagram while object declarations bind object names ($\overline{o}$) to class names ($\overline{c}$) and provide a set of field initialisations.

A class c is composed of a class' name $\overline{c}$, field (attribute) declarations, method (operation) declarations, and a single state machine μ. The type of a field can be a basic UML type (Integer, Boolean or String) or a reference type, *i.e.* , another class defined in the diagram. Fields can be static or non-static, as indicated by the value of a boolean flag *static*. In addition, static fields can be assigned default values. An operation m is defined by its name ($\overline{m}$) and list of parameter declarations $x_i : \tau_i$ denoting the parameter's name and type, respectively.

The class diagram of Figure 2 consists of two classes: Peer and Manager. Peer declares the fields peer, manager, share and role of type Peer, Manager, Boolean and Integer, respectively, and two operations, getRole and sendMsg. The operation getRole declares parameters r, s and p, of types Integer, Boolean and Peer, respectively, whereas sendMsg requires a single parameter, m of type String. Static fields are represented by over-lining the field's name and can optionally be assigned a default value. For example, the integer static fields J and P in class Manager are initialised with values 1 an 2, respectively.

The state machine μ of a class is composed of an initial state (s_0), a final state (s_f), and two finite sets of states and transitions. A state s ($s \in$ State) can be simple or composite. A composite state is made of substates grouped into regions. A composite state can be simple (SComposite), if it contains a single region, or orthogonal (OComposite), if it's subdivided into more than one region. All states and regions are labelled.

Figure 3 shows the state chart diagrams associated to the Peer and Manager classes of our example. The state machine diagram of class Peer is composed of the initial state P0, final state Pf and eight intermediate states. Initial states are represented by a full circle, fi-

$$
\begin{array}{rcl}
U \in \text{Model} & = & \mathcal{P}(\text{Class}) \times \mathcal{P}(\text{ObjectD}) \\
U & ::= & (c^*, O^*) \\
c \in \text{Class} & = & \text{Cname} \times \mathcal{P}(\text{Field}) \times \mathcal{P}(\text{Method}) \times \text{SM} \\
O \in \text{ObjectD} & = & \text{Oname} \times \text{Cname} \times \text{Action}_\perp \\
O & ::= & \overline{o} : \overline{c}\ \{\ a\ \}; \\
f \in \text{Field} & = & \text{Static} \times \text{Fname} \times \text{Type} \times \text{Constant}_\perp \\
m \in \text{Method} & = & \text{Mname} \times (\text{Var} \times \text{Type})^* \\
m & ::= & \overline{m}(x_1 : \tau_1, \ldots, x_n : \tau_n) \\
\tau \in \text{Type} & = & \text{Primitive} \cup \text{Reference} \\
\text{Primitive} & = & \{\mathsf{Integer}, \mathsf{Boolean}, \mathsf{String}\} \\
\text{Reference} & = & \text{Class} \\
 & & \\
\mu \in \text{SM} & = & \text{State} \times \text{State} \times \mathcal{P}(\text{State}) \times \mathcal{P}(\text{Trans}) \\
\mu & ::= & (s_0, s_f, S, T) \\
s \in \text{State} & = & \text{Simple} \cup \text{SComposite} \cup \text{OComposite} \\
s & ::= & \overline{s} \mid [\overline{s}, R] \mid [\overline{s}, R_1 \times \ldots \times R_n] \quad n > 1 \\
R \in \text{Region} & = & \text{Rname} \times \mathcal{P}(\text{State}) \\
R & ::= & (\overline{rg}, S) \\
tr \in \text{Trigger} & = & \text{Method} \\
t \in \text{Trans} & = & \text{Tname} \times \mathit{states}(\mu) \times \mathit{states}(\mu) \\
 & & \quad \times \text{Trigger}_\perp \times \text{Guard} \times \text{Action}_\perp \\
t & ::= & (\overline{t}, s_1, s_2, tr, g, a) \\
 & & \\
g \in \text{Guard} & g ::= & \mathsf{not}\ g \mid g\ \mathsf{and}\ g \mid g\ \mathsf{or}\ g \mid \mathsf{true} \mid \mathsf{false} \mid b \\
b \in \text{BPred} & b ::= & e_1\ op_b\ e_2 \mid \mathsf{tm}(n) \\
a \in \text{Action} & a ::= & \mathsf{call}\ x.\overline{m}(e_1, \ldots, e_n) \mid r = e \mid a_1; a_2 \mid \\
 & & \quad \mathsf{if}\ (b)\ a_1\ \mathsf{else}\ a_2 \\
e \in \text{AExp} & e ::= & e_1\ op_a\ e_2 \mid n \mid r \\
r \in \text{VarRef} & r ::= & x \mid f \mid r.f \mid c.f \mid \mathsf{this} \\
 & & \quad \text{where}\ \ x \in \text{Var}, n \in \mathit{Integer}
\end{array}
$$

Fig. 4 UML Model and Action Language

nal states by two concentric circles whereas intermediate states *e.g.* Branch, are represented by an oval surrounding the state's name. All state machines must have an initial and final state.

A transition $(\bar{t}, s_1, s_2, tr, g, a)$ is composed by its name $\bar{t}$, source state s_1 and target state s_2. A transition is graphically represented by an arrow joining its source and target states, together (optional) with a label indicating the transition's name. Additionally, a transition may carry annotations of the form $tr[b]/a$ to indicate the presence of trigger (tr), guard (g) and effect (a) elements. A trigger defines the event *i.e.* , an operation in the restricted form of UML that we assume in this paper, that triggers the execution of the transition. The guard defines the condition that must be satisfied in order for the transition to be executed. The effect specifies the action that is executed together with the change of state defined by the transition.

Guards and actions are left unspecified by UML in order to allow the user to adopt the notation that best suits the problem in hand. In USVF we fill this gap by adopting a specific notation for specifying guards and actions. This notation is introduced in the bottom part of Figure 4 (see last 5 definitions).

In particular, both guard expressions and actions contain references r to UML elements declared in the class diagram such as non-static fields (f and $r.f$), static fields ($c.f$) and operation parameter names x. The special variable this denotes the current object (also referred as self) *e.g.* f is equivalent to writing this.f. A guard g is a boolean expression that combines classic logical operators and constants with boolean predicates b, including the special timeout predicate tm(n) used to control the amount time spent by the machine in a particular transition. A transition annotated with a guard will only be executed if the guard is true. For example, a state machine of class Peer will only change from state Photo to P3 if the value of field share is false.

An action a can be a sequence of actions, an assignment, a call action or a conditional. Actions are executed every time the associated transition is triggered. For example, the transition that joins states R0 and R1 in the state machine of class Manager does not contain a trigger or guard and, therefore, is executed always regardless of the local state of the machine. The change of state, from R0 to R1, is accompanied by the execution of the call action associated to the transition, which sends a getRole message to the object stored by peer1. The message contains as parameters the values of static field J, non-static fields share and peer2, and the reference to self, this.

Now let's consider the transition joining states P0 and Branch in the state machine of class Peer. The transition contains the trigger geRole and, therefore, will only be executed when the state machine receives a message containing the getRole operation. When this happens, the values included in the message are bound to the formal parameters of getRole *i.e.* r, s, p and m, and the action associated to the transition is executed together with the change of state from P0 to Branch. In this case, the action is a sequence of assignments that updates the state machine's fields *e.g.* manager=m updates the value of manager to the value mapped to m sent in the message.

A model is completed, or closed, by adding a set of object declarations to the class and state machine diagrams. An object declaration $\overline{o} : \overline{c}\ \{\ a\ \}$; indicates that the system will be instantiated with an object $\overline{o}$ of class $\overline{c}$. Given the absence of object constructors, the action a is used to initialise the fields of the object. If a field is left uninitialised, the system assigns the default value specified in the class diagram or the default value associated to its type, in that order. In our example, we need to instantiate one Manager object and two Peer objects with initial values that satisfy certain constraints.

```
Peer a1;
Peer a2;
Manager manager {
        peer1=a1; peer2=a2; share=false; };
```

The object initialisation shown above is used to instantiate a "valid" initial configuration (section 2.3.4) of the model. The manager object must know in advance, as specified by the system, the locations of the other two objects. Therefore, the values of a1 and a2 must be assigned to the fields peer1 and peer2 right after object creation. The system also requires another input, namely, the value of the share field, which indicates if the peers can exchange information during the execution of their tasks. The USVF does not support a way of making these constraints explicit.

2.2 States and State Hierarchy

The definition of composite states (Figure 4) is, essentially, a tree structure with nodes made of composite states and regions (composite states branch into regions and regions branch into substates) and leafs made of simple states.

The set S in $\mu = (s_0, s_f, S, T)$ contains only the states present at the top level of the state machine, each defining a separate tree of states. In order to work with a single well-formed tree, we complete the tree structure induced by state machine μ by introducing a fresh root state that branches to all top-level states.

Definition 1 (State Machine Tree)
Given state machine $\mu = (s_0, s_f, S, T)$ and operations *substates*, *parent* and *parentR*, the state machine tree of μ is generated by adding the special *sroot* element such that:

$$
\begin{aligned}
substates(root) &= S \cup \{s_0, s_f\} \\
substates(s) &= \phi, s \in \text{Simple} \\
substates([\overline{s}, (\overline{rg}, S)]) &= S \\
substates([\overline{s}, R^*]) &= \textstyle\bigcup_{(rname,S)\in R^*} S \\
parent(s) &= s', s \in substates(s') \\
parentR(s) &= R, s \in R \ \wedge\ R \in parent(s)
\end{aligned}
$$

where operators *substates*, *parent* and *parentR* extract a state's substates, parent state (superstate) and parent region, respectively.

All states in a state machine tree are reachable from *root*. Thus, the complete sets of states in μ can be obtained by traversing the whole tree (starting from *root*) and collecting all the visited states.

Definition 2 (Substates of a State Machine) The complete set of states of state machine $\mu = (s_0, s_f, S, T)$ is defined by:

$$states(\mu) = \{s_0, s_f\} \cup states(root)$$
$$states(s) = \bigcup_{s \in substates(s)} \cup \{s\}$$

Definition 3 (State and Transition Ordering) The tree structure in state machine μ defines the partial order $(states(\mu), \preceq_\mu, \bot)$ where $\bot = root$ and $\preceq_\mu$ is defined as follows:

$$s \preceq_\mu s' \Leftrightarrow s' \in states(s)$$

where $\prec_\mu$ is the non-reflexive version of $\preceq_\mu$. We extend the order relation to transitions. Let s and s' be the source states of transitions t and t', respectively. We write $t \preceq t'$ if and only if $s \preceq s'$.

The definition above entails the following:

- $root \preceq s$, for all $s \in states(\mu)$.
- For all states $s' \in tpath(s)$, $root \preceq s' \preceq s$.
- A state s is reachable from s' if and only if $s' \preceq s$.
- The greatest lower bound of a set of states S ($\sqcap S$) defines the point where all paths $tpath(s)$, $s \in S$, join.

States are uniquelly identified by their fully qualified names (QSname). The fully qualified name of s - $name(s)$ - has two parts: a prefix, made of the concatenation of the names of the states and regions needed to traverse in order to get from *root* to s i.e. in $tpath(s)$, and the state's name $\overline{s}$.

$$name : \text{State} \rightarrow \text{QSname}$$
$$name(s) = prefix(s)\overline{s}$$
$$prefix(root) = \epsilon$$
$$prefix(s) = \begin{cases} prefix(s').\overline{s}' & s' \in \text{SComposite} \\ prefix(s').\overline{s}'.\overline{rg}' & s' \in \text{OComposite} \end{cases}$$
$$\text{where } s' = parent(s) \ \wedge \ (\overline{rg}', -) = parentR(s)$$

Region names are required only when accesing a substate in a composite ortogonal state whereas region names inside simple composite states are omitted.

2.3 UML Semantics

In this section, we define the operational semantics of UML models used in USVF. More specifically, we define the execution of UML models as a two-level operational semantics. Execution of the UML models considered in this paper is determined by two behavioural components: state machines and actions. The top-level semantics defines the execution of state machines and the interaction between the objects of the model. The low-level or action semantics defines the execution of the actions associated to machine transitions.

$$\begin{aligned} v \in \text{Value} &= \text{PrimVal} \cup \text{ObjRef} \\ \text{PrimVal} &= \text{Int} \cup \text{Bool} \cup \text{String} \\ \hat{o} \in \text{ObjRef} &= \text{Oname} \cup \{null\} \\ o \in \text{Object} &= \text{Class} \times (\text{Fname} \rightarrow \text{Value}) \\ H \in \text{Heap} &= (\text{ObjRef} \rightarrow \text{Object}) \ \cup \\ &\quad (\text{Cname} \times \text{Fname} \cup \text{Globals} \rightarrow \text{Value}) \\ e \in \text{Exp} &= \text{Guard} \cup \text{AExp} \\ \rho \in \text{LEnv} &= \text{Var} \rightarrow \text{Value} \\ \sigma \in \text{Env} &= \text{Heap} \times \text{Object} \times \text{LEnv} \qquad \sigma ::= \langle H, \hat{o}, \rho \rangle \\ \Downarrow \in \text{Env} \times \text{Exp} &\rightarrow \text{Value} \qquad \sigma, e \Downarrow v \end{aligned}$$

Fig. 5 Values and Expression Evaluation

To define this semantics, in Section 2.3.1, we introduce the set of values used by our semantics and define how expressions are evaluated. We continue in section 2.3.2 by defining the semantics of action execution. Finally, in section 2.3.4, we introduce the concepts of state machine and model configurations and provide the rules that define the execution of UML models.

2.3.1 Values and Expressions

Our execution model deals with two types of values, primitive values and object references (Figure 5). Primitive values correspond to primitive types and, therefore, can be integers, boolean constants $\{0, 1\}$, and strings. Object references $\hat{o}$ denote locations pointing to objects (o) in the heap H. The heap or global store is a mapping from static fields to values, and object references to objects. An object is a mapping from (non-static) fields and time variables (timer) to values instantiated by functions *new*, defined as follows:

$$\begin{aligned} new(H, c) = (H[\hat{o} \mapsto o], \hat{o}) \text{ where} \\ c = (\overline{c}, \underline{f}^*, -, -) \ \wedge \ \hat{o} = \overline{c} \wedge \\ o = \{(\bar{f}, v) \mid (0, \bar{f}, \tau, v) \in \underline{f}^*\} \cup \\ \{(\bar{f}, default(\tau)) \mid (0, \bar{f}, \tau, \bot) \in f^*\} \end{aligned}$$

where

$$default(\tau) = \begin{cases} null & \tau \in \text{Reference} \\ \epsilon & \tau = \mathsf{String} \\ 0 & \tau \in \{\mathsf{Integer}, \mathsf{Boolean}\} \end{cases}$$

Function *new* assigns new location $\hat{o}$ to newly allocated object o. Fields are initialised to default values. In particular, fields of reference type are set to the special *null* reference. Similarly, the mapping of static fields of a class is initialised as follows:

$$\begin{aligned} static(c) = \{(\overline{c}, \bar{f}, v) \mid (1, \bar{f}, \tau, v) \in \underline{f}^*\} \cup \\ \{(\overline{c}, \bar{f}, default(\tau)) \mid (1, \bar{f}, \tau, \bot) \in f^*\} \\ \text{where } c = (\overline{c}, f^*, -, -) \end{aligned}$$

$msg \in \text{Message} \quad \varepsilon \in \text{Event} \quad q, iq, oq \in \text{Q} \quad \alpha \in \mathcal{P}(\text{Event})$

$$
\begin{aligned}
msg &::= o_1, o_2, m(v_1, \ldots, v_n) \\
\varepsilon &::= \mathsf{send}(msg) \mid \mathsf{recv}(msg) \mid \mathsf{msg}(msg) \mid \\
&\quad\ \mathsf{write}(o, c, f) \mid \mathsf{trans}(o, t) \\
q &::= msg^*
\end{aligned}
$$

$$
\begin{aligned}
\xrightarrow{\alpha,q} &\in \text{Env} \times \text{Action}_{\perp} \times \mathcal{P}(\text{Event}) \times \text{Q} \times \text{Env} \\
\gamma \in \text{SMConf} &= \text{Object} \times \text{SM} \times \widehat{\text{State}} \times \text{Q} \times \text{Q} \\
\Gamma \in \text{Conf} &= \text{Heap} \times \mathcal{P}(\text{SMConf}) \\
\hat{s} \in \widehat{\text{State}} &= \mathcal{P}(\text{Simple}) \\
\xrightarrow{\alpha} &\in \text{Model} \times \text{Conf} \times \mathcal{P}(\text{Event}) \times \text{Conf}
\end{aligned}
$$

$$
\begin{aligned}
&\Gamma ::= \langle H, \{\gamma_1, \ldots, \gamma_n\}\rangle \qquad \sigma, a \xrightarrow{\alpha,q} \sigma' \\
&\gamma ::= \langle o, \mu, s, iq, oq\rangle \qquad \langle H, \{\gamma_1, \ldots, \gamma_n\}\rangle \xrightarrow{\alpha} \Gamma'
\end{aligned}
$$

Fig. 6 UML Model - Semantic Domains

An environment $\sigma = \langle H, \hat{o}, \rho\rangle$ keeps track of all the variable bindings valid at a particular program point, including global variables and static fields stored in H and the non-static fields of the current object $\hat{o}$. The local environment ρ contains the binding generated by the execution of a transition e.g. parameters contained in the trigger. Environment look-up, denoted by $\mathcal{L}(\sigma, r)$, where $\sigma = \langle H, \hat{o}, \rho\rangle$ is performed as follows:

$$
\begin{aligned}
&\mathcal{L}(\sigma, c.f) = H(c.f) \qquad \mathcal{L}(\sigma, f) = H(\hat{o})(f) \\
&\mathcal{L}(\sigma, x.f) = \mathcal{L}(\sigma, x)(f) \qquad \mathcal{L}(\sigma, x) = \rho(x) \\
&\mathcal{L}(\sigma, \mathsf{this}) = \hat{o}
\end{aligned}
$$

Boolean ($g \in$ Guard) and arithmetical ($e \in$ AExp) expressions evaluate to values. Expression evaluation is denoted by $\sigma, e \Downarrow v$, where the evaluation function $\Downarrow$ takes expression e and environment σ and returns value v.

Let $\sigma, g_i \Downarrow v_i$ and $\sigma, e_i \Downarrow v_i$. The evaluation of boolean expressions is defined as follows:

$$
\begin{aligned}
&\sigma, \mathsf{true} \Downarrow 1 \qquad \sigma, (\mathsf{not}\ g) \Downarrow (\neg v) \\
&\sigma, \mathsf{false} \Downarrow 0 \qquad \sigma, (g_1\ \mathsf{and}\ g_2) \Downarrow (v_1 \wedge v_2) \\
&\sigma, \mathsf{null} \Downarrow null \qquad \sigma, (g_1\ \mathsf{or}\ g_2) \Downarrow (v_1 \vee v_2) \\
&\sigma, (e_1\ op_b\ e_2) \Downarrow \|op_b\|(v_1, v_2)
\end{aligned}
$$

where $\|op_b\|$ is the predicate associated to operator op_b. Similarly, evaluation of arithmetical expressions is defined by:

$$\sigma, n \Downarrow n \quad \sigma, r \Downarrow \mathcal{L}(\sigma, r) \quad \sigma, (e_1\ op_a\ e_2) \Downarrow \|op_a\|(v_1, v_2)$$

Our semantics is equipped with a set of time variables (timers) used to keep track of the number of steps executed by the model. We define a global clock tm and a set timers $tm(\overline{o}, s)$ - one for each state in every object declared in the model - together with the following operations:

$$
\begin{aligned}
&tm \in \text{Globals},\ tm(\overline{o}, s) \in \text{Globals}, \forall \overline{o}, s \in U \\
&H^{\surd} = H[tm \mapsto x], x = H(tm) + 1 \\
&H^{\surd o, s^*} = H[tm \mapsto x][tm(\overline{o}, s) \mapsto x]_{s \in s},
\end{aligned}
$$

where $H^{\surd}$ increments the global clock by one and $H^{\surd o, s*}$ increments the clock and sets the timers of each state in s^* to the new clock value.

The $\mathsf{tm}_s(n)$ guard checks if the time elapsed since state s was entered is greater or equal then n:

$$\langle H, \hat{o}, \rho\rangle, \mathsf{tm}_s(n) \Downarrow (H(tm) - H(tm(name, s))) \geq n$$

where s is the source state of the transition labelled by $\mathsf{tm}(n)$

2.3.2 Action Semantics and Events

Figure 6 defines the semantic domains and runtime structures involved in the execution of UML models. More specifically, it defines the domain of the execution relations $\xrightarrow{\alpha,q}$ and $\xrightarrow{\alpha}$ used to model the execution of actions (low-level semantics) and UML state machines (top-level semantics), respectively.

State machine actions (Section 2.1, Figure 4) modify an object's state (fields) and generate events captured by the top-level semantics. An action is executed when the transition associated with it is scheduled by the top-level semantics. Action execution generates two kinds of events:

- $\mathsf{send}(o_1, o_2, m(v_1, \ldots, v_n))$, generated by the execution of a call action.
- $\mathsf{write}(o, c, f)$, generated by the execution of an assignment. It reports the modification of field f on object o of class c.

The rest of the events i.e recv, msg and trans; are generated by the top-level semantics as explained in section 2.3.4.

Definition 4 (Action Semantics) Action execution is defined as the smallest relation $\longrightarrow$ that satisfies the rules in Figure 7. We write $\sigma, a \xrightarrow{\alpha,q} \sigma'$ to denote the execution of action a under environment σ, where α denotes the set of generated events, q the queue of messages to be sent, and σ' reflects the changes made to σ after the execution of a.

Rule (7.1) shows the execution of the empty action $\perp$, used to express the absence of an action in a transition. Rule (7.2) implements sequencing, that is, the composition of the execution of actions a_1 and a_2.

Message passing, defined by rule (7.3), is started by the execution of action call $r.m(e_1, \ldots, e_n)$. The execution of call creates a message containing the current object's location $\hat{o}$, the target $\hat{o}'$ obtained from reference r, the method's name and the evaluated arguments. The message, $(\hat{o}, \hat{o}', m(v_1, \ldots, v_n))$, is used to generate a new send event and placed into the action's message queue.

Rules (7.4) and (7.5) implement variable assignment. They contemplate two cases: the first case updates the value of static fields and local variables while the second case deals with field update. Field update modifies the heap by updating the target object and generates a write event indicating the field modified by the action.

(7.1) $$\langle H, \hat{o}, \rho\rangle, \bot \xrightarrow{\phi,\epsilon} \langle H, \hat{o}, \rho\rangle$$

(7.2) $$\frac{\langle H, \hat{o}, \rho\rangle, a_1 \xrightarrow{\alpha_1,q_1} \langle H_1, \hat{o}, \rho_1\rangle \quad \langle H_1, \hat{o}, \rho_1\rangle, a_2 \xrightarrow{\alpha_2,q_2} \langle H', \hat{o}, \rho'\rangle \quad \alpha = \alpha_1 \cup \alpha_2 \ \wedge \ q = q_1 :: q_2}{\langle H, \hat{o}, \rho\rangle, a_1; a_2 \xrightarrow{\alpha,q} \langle H', \hat{o}, \rho'\rangle}$$

(7.3) $$\frac{\hat{o}' = \mathcal{L}(\langle H, \hat{o}, \rho\rangle, r) \quad i = 1, \ldots, n : \langle H, \hat{o}, \rho\rangle, e_i \Downarrow v_i \quad msg = (\hat{o}, \hat{o}', m(v_1, \ldots, v_n)) \quad \alpha = \{\mathsf{send}(msg)\}}{\langle H, \hat{o}, \rho\rangle, \mathsf{call}\ r.m(e_1, \ldots, e_n) \xrightarrow{\alpha,msg} \langle H, \hat{o}, \rho\rangle}$$

(7.4) $$\frac{\langle H, \hat{o}, \rho\rangle, e \Downarrow v \quad (H', \rho') = \begin{cases} (H, \rho[x \mapsto v]) & r = x \\ (H[c.f \mapsto v]) & r = c.f \end{cases}}{\langle H, \hat{o}, \rho\rangle, r = e \xrightarrow{\phi,\epsilon} \langle H', \hat{o}, \rho'\rangle}$$

(7.5) $$\frac{\langle H, \hat{o}, \rho\rangle, e \Downarrow v \quad \hat{o}' = \begin{cases} \mathcal{L}(\langle H, \hat{o}, \rho\rangle, x) & r = x.f \\ \hat{o} & r = f \end{cases} \quad \alpha = \{\mathsf{write}(\hat{o}', f)\} \quad H' = H[\hat{o}' \mapsto H(\hat{o}')[f \mapsto v]]}{\langle H, \hat{o}, \rho\rangle, r = e \xrightarrow{\alpha,\epsilon} \langle H', \hat{o}, \rho\rangle}$$

Fig. 7 Action Semantics

2.3.3 State Trees and Tree Rewrite

One of the main consequences of having composite states is that, at any point during execution of the model, state machines may have more than one active (current) state. Not only does the existence of orthogonal composite states spawn a set of concurrent active states. Given leaf state s, the set of all states in $tpath(s)$ also become active *i.e.* a transition leaving any superstate of s can potentially be fired.

Thus, the set of active states form also a tree structure $\hat{s} \in \widehat{\text{State}}$. We call $\hat{s}$ a state tree and represent it by listing all its leaf states. In this way, tree trimming and extension can be implemented with set operations.

State trees are initialised by the *start* function:

$$\begin{aligned} start(s) &= \{s\}, s \in \text{Simple} \\ start([\overline{s}, R]) &= \{initialR\} && \text{(SComposite)} \\ start([\overline{s}, R^*]) &= \{initialR \mid R \in R^*\} && \text{(OComposite)} \end{aligned}$$

State trees are transformed by the execution of transitions. Such transformation, state tree rewrite, is defined by $\hat{s} \overset{t}{\rightsquigarrow} \hat{s}'$.

Definition 5 (State Tree Rewrite) Given state tree $\hat{s} = \{s_1, \ldots, s_n\}$ and transition $t = (s, s', -, -, -)$, we define $\hat{s} \overset{t}{\rightsquigarrow} \hat{s}'$ ($\hat{s}$ is transformed into $\hat{s}'$ by applying t) as follows:

$$\begin{aligned} \hat{s} \overset{t}{\rightsquigarrow} \hat{s}' \Leftrightarrow\ & t = (-, s, s', -, -) \\ & \hat{s} = S \cup S' \ \wedge \ \hat{s}' = S' \cup start(s') \\ & s_x = s \sqcap s' \\ & S = \{s_l \in \hat{s} \mid s_l \in s_x\} \end{aligned}$$

2.3.4 UML Operational Semantics

The operational semantics of UML models is defined as a *small step semantics* [39] on program configurations. A program configuration $\Gamma = \langle H, \{\gamma_1, \ldots, \gamma_n\}\rangle$ represents the state of execution, at any given time, of a UML model. It is made of the heap H and the set of machine configurations keeping runtime information for each of the objects declared in the model. A machine configuration $\gamma = \langle \hat{o}, \mu, \hat{s}, iq, oq\rangle$ is made of the object's location $\hat{o}$, the state machine μ declared for the object's class, the machine's current state s, and the input and output queues, iq and oq, respectively.

Definition 6 (Initial Program Configuration) Let $U = (c^*, O^*)$. An initial program configuration of model U is the configuration obtained after the initialisation of all the objects declared in O^*. We say that Γ is an initial configuration of $U = (c^*, O^*)$, and write $U \vdash^{\text{I}} \Gamma$, if initialisation takes place as follows:

$$\begin{aligned} & (c^*, O^*) \vdash^{\text{I}} \langle H, \{\gamma_1, \ldots, \gamma_n\}\rangle \Leftrightarrow \\ & \quad H = H_n \ \wedge \ O^* = \{O_1, \ldots, O_n\} \wedge \\ & \quad i \in \{1, \ldots, n\} : \\ & \qquad \gamma_i = \langle \hat{o}_i, \mu_i, start(s_i), \epsilon, \epsilon\rangle \ \wedge \ c_i = (-, -, -, \mu_i) \\ & \qquad O_i = \overline{o}_i : \overline{c}_i \ \{\ a_i\ \}; \\ & \qquad \langle H_{i-1}, \hat{o}_i, \rho\rangle, a_i \xrightarrow{\phi,\epsilon} \langle H_i, \hat{o}_i, \rho\rangle \\ & \text{where} \\ & \quad \rho = [\overline{o}_i \mapsto \hat{o}_i]_{i \in \{1,\ldots,n\}} \ \wedge \ H_0 = H'_m \\ & \quad c^* = \{c_i, \ldots, c_m\} \\ & \quad j \in \{1, \ldots, m\} : (H'_j, \hat{o}_j) = new(H_{j-1}, c_j) \\ & \quad H'_0 = (\textstyle\bigcup_{c_j \in c^*} static(c_j)) \cup \text{Globals}(U) \end{aligned}$$

In other words, initialisation takes place in the following order:

- All static fields and global variables (clock and timers) are initialised and placed in the initial heap H'_0.
- All objects declared in O^* are allocated and initialised with default values (*new* function defined in 2.3.1), generating heap H'_m and local environment ρ..
- All initalisation actions a_i are executed, resulting in final heap $H = H_n$.
- A state machine configuration γ_i is generated per declared object O_i, where $start(s_i)$ is the initial state tree of μ_i.
- The resulting heap is paired with the set of machine configurations *i.e.* $\langle H, \{\gamma_1, \ldots, \gamma_n\}\rangle$.

State machine execution is driven by transition execution. We say that a transition is enabled if its source

$$(8.1)\quad \frac{\begin{array}{c}\gamma_1 = \langle o_1, \mu_1, \hat{s}_1, iq_1, msg : oq_1\rangle \ \wedge\ \gamma_2 = \langle o_2, \mu_2, \hat{s}_2, iq_2, oq_2\rangle \\ msg = (-, o_r, m(-)) \ \wedge\ (o_r = o_2) \ \wedge\ \alpha = \{(\mathsf{msg}\ \ msg)\} \\ \gamma_1' = \langle o_1, \mu_1, \hat{s}_1, iq_1, oq_1\rangle \ \wedge\ \gamma_2' = \langle o_2, \mu_2, \hat{s}_2, iq_2, oq_2 : msg\rangle\end{array}}{U \vdash \langle H, \{\gamma_1, \gamma_2\} \cup \gamma^*\rangle \xrightarrow{\alpha} \langle H, \{\gamma_1', \gamma_2'\} \cup \gamma^*\rangle^{\surd}}$$

$$(8.2)\quad \frac{\begin{array}{c}t \in enabled(H, \gamma) \ \wedge\ \gamma = \langle o, \mu, \hat{s}, iq, oq\rangle \ \wedge\ t = (-, -, \bot, g, a) \ \wedge\ \hat{s} \overset{t}{\rightsquigarrow} \hat{s}' \\ \langle H, o, \bot\rangle, a \xrightarrow{\alpha', q} \langle H', o, \bot\rangle \ \wedge\ \alpha = \alpha' \cup \{\mathsf{trans}(o, t)\} \ \wedge\ \gamma' = \langle o, \mu, \hat{s}', iq, oq :: q\rangle\end{array}}{U \vdash \langle H, \{\gamma\} \cup \gamma^*\rangle \xrightarrow{\alpha} \langle H', \{\gamma'\} \cup \gamma^*\rangle^{\surd o}}$$

$$(8.3)\quad \frac{\begin{array}{c}t \in enabled(H, \gamma) \ \wedge\ \gamma = \langle o, \mu, \hat{s}, msg : iq, oq\rangle \ \wedge\ t = (-, -, m, g, a) \\ msg = (-, o, m(v_1, \ldots, v_n)) \ \wedge\ M = m(x_1, \ldots, x_n) \ \wedge\ \rho = [x_i \mapsto v_i] \\ \hat{s} \overset{t}{\rightsquigarrow} \hat{s}' \ \wedge\ \langle H, o, \rho\rangle, a \xrightarrow{\alpha', q} \langle H', o, -\rangle \ \wedge\ \alpha = \alpha' \cup \{\mathsf{recv}(msg), \mathsf{trans}(\hat{o}, t)\}\end{array}}{U \vdash \langle H, \{\gamma\} \cup \gamma^*\rangle \xrightarrow{\alpha} \langle H', \{\langle o, \mu, \hat{s}', iq, oq :: q\rangle\} \cup \gamma^*\rangle^{\surd o}}$$

$$(8.4)\quad \frac{enabled(H, \gamma) = \phi \ \wedge\ hasTriggers(H, \gamma) \neq \phi \ \wedge\ \gamma = \langle o, \mu, \hat{s}, msg : iq, oq\rangle}{U \vdash \langle H, \{\gamma\} \cup \gamma^*\rangle \xrightarrow{\phi} \langle H, \{\langle o, \mu, \hat{s}, iq, oq\rangle\} \cup \gamma^*\rangle^{\surd}}$$

$$(8.5)\quad \frac{\forall \gamma \in \gamma^*.(enabled(H, \gamma) = \phi \ \wedge\ \gamma = \langle -, -, -, iq, \epsilon\rangle)}{U \vdash \langle H, \gamma^*\rangle \xrightarrow{\phi} \langle H, \gamma^*\rangle^{\surd}}$$

Fig. 8 UML Operational Semantics

state is part of the current state of the machine configuration and its firing conditions are satisfied. Only one enabled transition will be executed at the time.

We separate transitions into two groups, completion (no trigger) and triggered transitions. Transition firing conditions are checked by the $completion(H, \gamma, s)$ and $triggered(H, \gamma, s)$, which return the set of enabled completion and triggered transitions, respectively, leaving source state s.
A completion transition is enabled if its guard evaluates to true:

$$completion(H, \langle o, \mu, \hat{s}, iq, oq\rangle, s) = \{t \in trans(\mu).\ t = (s, -, \bot, g, -) \ \wedge\ \langle H, o, \bot\rangle, g \Downarrow 1\ \}$$

Triggered transitions are enabled only if the required trigger (operation) is found at the front of the object's input queue and its guard evaluates to true:

$$triggered(H, \langle o, \mu, \hat{s}, msg : iq, oq\rangle, s) = \{t \in trans(\mu).\ t = (s, -, m, g, -) \ \wedge\ msg = (-, o, m(v_1, \ldots, v_n)) \wedge M = m(x_1, \ldots, x_n) \ \wedge\ \rho = [x_i \mapsto v_i] \wedge \langle H, o, \rho\rangle, g \Downarrow 1\ \}$$

Given current (leaf) state s, a transition is enabled if its firing conditions are satisfied and s is reachable (substate) from the transition's source state s' *i.e.* $s' \preceq s$. It may be the case that the set of enabled transitions contains conflicting transitions, that is, transitions t and t' where the source state of one of them is a substate of the other *e.g.* $t \prec t'$. If that is the case then transtions with higher order (states deeper in the tree) should be given priority *e.g.* t'. Furthermore, completion states should be given priority against triggered transitions when calculating the set of enabled transitions leaving the same state.

The set $enabled(H, \gamma, s)$ of enabled transitions associated to a current single state s is defined as follows:

$$enabled(H, \gamma, s) = \begin{cases} \phi & s = root \\ enabledL(H, \gamma, s) & enabledL(H, \gamma, s) \neq \phi \\ enabled(H, \gamma, parent(s)) & \text{otherwise} \end{cases}$$

$$enabledL(H, \gamma, s) = \begin{cases} completion(H, \gamma, s) & completion(H, \gamma, s) \neq \phi \\ triggered(H, \gamma, s) & \text{otherwise} \end{cases}$$

Finally, the set of enabled transitions associated to a state machine configuration $\gamma = \langle o, \mu, \hat{s}, iq, oq\rangle$ is defined as follows:

$$enabled(H, \gamma) = \bigcup_{s \in \hat{s}} enabled(H, \gamma, s)$$

We are now ready to define the operational semantics of state machines.

Definition 7 (UML Small Step Semantics) Let $\longrightarrow$ be the smallest relation that satisfies the rules in Figure 8. We write $U \vdash \Gamma \xrightarrow{\alpha} \Gamma'$ to denote the execution of one computational step from program configuration Γ to Γ'. The change of configuration may generate a set of runtime events, denoted by α.

The rules in Figure 8 are mainly concerned with message passing and the execution of transitions. Message passing is realised in three steps. First, the sender executes a call action which places the message in the sender's output queue. Second, when the message reaches the front of the queue, the scheduler removes it and places it at the back of the receiver's input queue. Third, the message is removed from the top of the receiver's input

queue when, and if, there are only triggered transitions to execute.

Rule (8.1) describes the role of the scheduler. It removes the message from the output queue of o_1 in state machine configuration γ_1, matches the recipient identity with o_2 and places the message in the recipient's input queue. This step generates event msg(msg).

Rule (8.2) describes the execution of completion transitions. The execution of this rule generates a trans event, together with the events α' generated by the execution of the action a attached to the transition.

Rule (8.3) describes the execution of a triggered transition in state machine configuration γ. If the message msg in front of the input queue of γ matches the trigger of the transition (and the guard evaluates to true), the associated action is executed and the state machine configuration changes its current state. Note that the evaluation of the guard and the execution of the action (and the evaluation of the guard, performed by *enabled*) uses the values passed as arguments in the message by creating a new local environment $\rho = [x_i \mapsto v_i]$. This rule generates recv and trans events, as well as the ones generated by the action a associated to the transition.

If there no enabled transitions but there are triggered transitions (with source state in $\hat{s}$'s path) waiting for messages then rule (8.4) is executed. This means that deferred events are not considered i.e. events that do not trigger any transitions are discarded. In rule (8.4), if no enabled transitions t can be found and $hasTriggers(H, \gamma)$ is not empty, message msg is removed from the front of the input queue of state machine configuration γ.

If any of the conditions required by the rules above are satisfied *i.e.* no transitions are enabled and all output queues are empty, then rule (8.5) is fired.

All the rules increment the global clock tm by one by executing the $^{\surd}$ operation on the resulting configuration. If there is a change of state - rules (8.2)-(8.3) - the timers of the new states are set to the new clock value:

$$\langle H, \gamma^* \rangle^{\surd} = \langle H^{\surd}, \gamma^* \rangle$$
$$\langle H, \gamma^* \rangle^{\surd o} = \langle H^{\surd o, \hat{s}} \rangle \text{ where } \langle o, \mu, \hat{s}, -, - \rangle \in \gamma^*$$

2.3.5 The Example

Let's go back to the model defined by the class and state chart diagrams shown in Figures 2 and 3. The model defines two classes, Peer and Manager. A Manager object requires as input two peer objects, which must be assigned to fields peer1 and peer2. Therefore, a correct instantiation of the model should contain a Manager object and two Peer objects with the correct initialisations. As noted in section 2.1 we complete our model by writing:

```
Peer a1;
Peer a2;
Manager manager {
      peer1=a1; peer2=a2; share=false; };
```

After all initialisations and object allocations are finished, the program configuration contains three state machine configurations, all set to the machines' initial states. We now describe the steps taken by each of the objects.

A Peer object always starts by blocking on its initial state, waiting for the arrival of message getRole. Execution will proceed only when a getRole message reaches the front of the input queue. When this happens, the arguments sent with the message are assigned to variables p, m, r and s, respectively, and the action peer=p; ... is executed. Once the information regarding the assigned task (manager,role, shared and peer) is stored in the respective fields, a peer branches depending on its role. If the value of field shared is true, the Journalist object sends a sendMsg operation to its sibling and the Photographer object blocks and waits for the sendMsg operation to reach the front of the input queue. Once the message is dequeued, the Photographer proceeds to state Join. If the value of shared is false, both peers go straight to state Join. At this point, a peer reports the completion of the assigned task by sending an endTask message to the manager, sends a sendMsg to its sibling peer and waits until the a similar message arrives from its sibling peer.

The Manager object executes four transitions. The first transition contains an action which, when executed, sends a getRole message to peer1 containing the peer's role (Manager.J), information indicating if the peers can communicate during the execution of the assigned task (shared), the location of its sibling (peer2) and its own location (this). The second transition does the same for peer2 and role Manager.P. Finally, the manager waits for the arrival of two endTask messages from the peers indicating the completion of the news coverage event.

In this example, the internal execution of each state machine is deterministic. However, the execution of the whole system is not. Transition execution and message passing can interleave thus generating several execution traces or paths. The following section formalises the notion of execution path and defines a language used to verify properties against all possible execution traces. We will find that, for our example, not all execution traces fit the intended behaviour.

3 Property Specification and Verification

3.1 The USVF Property Specification Language

LTL is a popular formalism well suited not only for the verification of general system requirements, but also for the specification of security properties. However, in order to be useful in the context of UML models, LTL has to be able to explicitly reason about transition execution, states, class values and messages. In the following we introduce the USVF property specification languageas

$$
\begin{aligned}
\Phi &::= op^1\ \Phi \mid \Phi\ op^2\ \Phi \mid P \\
op^1 &= \{\mathsf{not}, \mathsf{next}, \mathsf{always}, \mathsf{eventually}\} \\
op^2 &= \{\mathsf{until}, \mathsf{and}, \mathsf{or}, \mathsf{implies}\} \\
P &::= P^b \mid P^e[L] \\
P^b &::= b \mid \mathsf{state}(r, s) \\
P^e &::= \mathsf{trans}(r, \bar{t}) \mid \mathsf{write}(r, \bar{f}) \mid \\
&\quad\ \mathsf{send}(r_1, r_2, m) \mid \mathsf{recv}(r_1, r_2, m) \mid \mathsf{msg}(r_1, r_2, m) \\
L &::= .\mathsf{and}\{g\} \mid .\mathsf{implies}\{g\} \\
&\text{where } b \in \mathrm{BPred}, g \in \mathrm{Guard}, s \in \mathrm{QSname}
\end{aligned}
$$

Fig. 9 Property Specification Language

an extension of LTL that tackles these problems and in Section 3.4 we formally define its semantics.

LTL reasons about the validity of predicates over all execution traces of a model. The syntax of the formulae Φ used to specify properties of the execution of UML models is defined by the grammar shown in Figure 9. According to this grammar, a formula Φ in the USVF property specification language is made of binary and unary temporal and logical operators applied recursively on local predicates P. The LTL operators used by USVF are always, eventually and until. For example, we can write

P1 always (o1.value < 100)
P2 eventually (o2.balance >= Account.Limit)
P3 (o3.balance < 500) until (o3.overdraft=true)

where P1 is true if the field value of object o1 is less than 100 on every execution state, P2 is true if the field balance of object o2 becomes, at some point, equal to the value of static field Account.Limit, and P3 is true if the value of field balance in object o3 does not exceed 500 in the states previous to field overdraft becoming true.

A predicate P can either be state predicate P^b, which expresses properties about the state of the system (*e.g.* objects, static and non-static fields), and a machine predicate P^e, which expresses properties about the effects and events generated by the execution of state machines (*e.g.* actions, transitions and message passing). In P^b, we re-use the set of predicates b used in the specification of UML models (Figure 4) and add the predicate state where state(o,s) checks if the current state of object o is s.

P^e can be one of the special predicates send, recv, msg, write and trans specific to UML state machines:

- send(o1,o2,m) checks if object o1 has made a call to operation m in object o2.
- recv(o1,o2,m) checks if object o2 has received *i.e.* removed from the input queue, a message from o1 containing operation m.
- msg(o1,o2,o3) checks if the message has been sent by the scheduler.
- write(o,f) checks if field f has been modified in object o.
- trans(o,t) checks if transition t has been executed in object o.

For example, given the following formulas:

P4 eventually state(o1,S2)
P5 always (send(o1,o2,getValue) implies (eventually send(o2, o1, receiveValue))
P6 always (state(o2,R1) implies (o1.result > 100))
P7 always state(o1,initial) implies eventually trans(o2,R2)

P4 checks if the state machine of o1 eventually reaches state S2, P5 checks that all calls to getValue are matched by a call to receiveValue, P6 is true if the value of result in object o1 is always greater than 100 every time o2 reaches state R1, and P7 will check that transition R2 in o2 is always executed after (at some point in the future) o1 reaches state initial.

Of particular interest is the special (optional) scope construct L added to the machine predicates P^e. By writing send(o1,o2,m).and{ x < 2 } the user can reason about the arguments of operations. Assuming x is declared as argument of m, the predicate above will be true if there is a call of m from o1 to o2 and the value of x is less than 2. Furthermore, the scope construct brings the active object related to the predicate into scope. For example, all fields of object o are within the scope of L in state(o,s) and thus, can write state(o,s).and{ f < 5 } where f is a field of f.

The scope construct also allows the user to access the values of special system variables linked to the occurrence of certain events. These variables are SENDER, RECEIVER and METHOD, available for predicates send, recv, msg. For example, if we want to check that no calls are made to any method in object o from objects o1 or o2 we can write:

```
always msg(*,o,*).implies{SENDER != o1 and
                          SENDER != o2}
```

3.2 Specification of Security Properties

Availability, integrity and confidentiality are, essentially, special cases of liveness and safety properties. By extending LTL to handle explicitly UML elements we provide a basic framework to specify security properties of UML models. Schneider [36] provides a precise characterisation of the class of enforceable security properties, specified by security automata. The set of constructs provided the USVF specification language allows for the specification of such class of properties. For example, the property stating that the first call from o1 to o2 must be Read followed by no calls to Send can be specified by writing:

```
call(o1,o2,*).implies{METHOD != Read} until
(call(o1,o2,Read) and
(always call(o1,o2,*).and{METHOD != Send}))
```

Furthermore, attacker models can be specified by constructing state machines that implement malicious behaviour. For example, an impersonation attack can be

implemented by placing a state machine that: (1) simulates the behaviour of the intended recipient - using the recipient's state machine - (2) sends part of the messages to the original recipient and (3) introduces new behaviour. A denial of service attack can be implemented by creating a state machine that reads the messages from a particular sender and either drops messages or inundates the original sender with reply messages.

3.3 The Example

The USVF property specification language can be used to express desired properties that should be checked for the example system introduced earlier in the paper. For example, verification of:

eventually state(a1,Branch)

gets us back true. We also get a positive answer when we check for:

always (state(a1,Branch) implies
always a1.role=Manager.J)

which means that, after the object reaches the state Branch, the role field always contains the intended value. However, if we check:

eventually state(a2,Pf)

we get back an error. This means that at least one execution path does not satisfy the property. After close inspection of the counter-example reported by the model checker we find out that the object with role Journalist may get the sendMsg message before getRole arrives. The semantics instructs the state machine to consume the message and wait for getRole, which eventually arrives. The initial message is lost and the machine will get stuck at state Msg.

We solve the problem by adding a synchronisation variable count implemented as an static field of Peer. The variable is initialised to 0 and incremented by 1 when getRole is processed. We must also strenghten the guards leaving Branch. For example, the transition corresponding to the Photographer role should be guarded as follows:

[role=Manager.P and Peer.count=2]

Continuing with our example, we may want to check that all getRole invocations pass as argument the value of the Manager's field share:

always send(a1,a2,getRole).implies{ s = share }

Similar checks can be performed to ensure that R1 from Table 1 ("the Manager starts the process by assigning roles and task to logged-in peers") is satisfied. Requirement R2 *i.e.* "A peer can adopt a Journalist or Photographer role", can be verified with the following formula for a1:

always (state(a1,Branch) implies
(always (a1.role= Manager.J or a1.role=Manager.P))

Note, however, that including only the second half of the formula will give us an error since roles are assigned at state Branch. Requirement R3 is easily verified by first checking (in the class diagram) that the sendMsg operation satisfies the required signature and that the operation is actually invoked *e.g.* eventually msg(*,a1,sendMsg).

We now proceed to show how the requirements listed in Table 1 related to the security properties defined in section 1.2, namely *availability*, *integrity* and *confidentiality*, can be specified using the property specification language.

Availability, which deals with the readiness of a system to provide timely data and functionality, can be specified in several ways. For example, a state from the state machine diagram can be specified as a ready state which must eventually be reached by the system. Also, availability of operations can be specified by forcing an answer after an operation request. This is the case of requirement R4 where we want to make sure that all task assignments are matched by a endTask response:

always (send(manager,a2,getRole) implies
(eventually recv(a2,manager,multiply)))

Integrity is concerned with the unauthorised modification of an object's state. Integrity can be verified by ensuring that a particular sequence of operations or actions does or does not take place - as in [36] - or by verifying that write operations are not performed in a particular object during a specific situation. For example, if we want to check that the value of the role field does not change after state Branch - requirement R5 - we should write:

always (state(a1,Branch) implies
(always (not write(a1,role))))

Confidentiality (R6 and R7) is concerned with ensuring authorised access to data in a system. Requirement R6 stipulates that all role assignments should come from Manager. Thus, R6 for peer a1 can be specified as follows:

always recv(*,a1,getRole).implies{SENDER=manager}

The final requirement, R7, requires that all communication between the peers during the execution of the task must be authorised. Such, authorisation is determined by the value of field share. Then, we can check for the occurrence of invocations to getRole. However, if we write:

always (send(a1,a2,sendMsg)
implies (manager.share=true))

we will get an error since the peers do exchange messages after the completion of the assigned task. The correct way of specifying the property is to restrict the check to the states between Branch and End:

always (state(a1,Branch) implies
((send(a1,a2,sendMsg) implies
(manager.share=true))
until state(a1,End)))

3.4 Verification by Model Checking

The USVF Property Specification Language is an LTL-based language that deals with UML elements defined by

class and state machine diagrams. In a system of temporal logic, various temporal logic operators or modalities are provided to describe and reason about how the truth values of assertions vary with time. In our system, we want to reason about the execution of UML models as defined by the semantics presented in Section 2. In this section, we start by building the notion of UML execution trace using the definition of the execution relation (Definition 7) and use it to specify the semantics of the USVF Property Specification Language.

We represent the execution of a UML model with the set of all possible execution paths generated from all possible initial configurations. An execution path Λ is a sequence of states λ made of pairs (Γ, α). Let $\Lambda = (\lambda_0, \lambda_1, \lambda_2, \ldots)$. We write $Path(U, \Lambda)$ if Λ is a valid execution path of U:

$$\begin{aligned} Path(U,\Lambda) \Leftrightarrow \lambda_0 = (\Gamma_0, \alpha_0) \ \wedge \ U \vdash^{\mathrm{I}} \Gamma_0 \\ \forall i > 0.\, U \vdash \Gamma_{i-1} \xrightarrow{\alpha} \Gamma_i \end{aligned}$$

that is, if the first state corresponds to an initial configuration of U and each pair of adjacent states correspond to a computational step.

Definition 8 (Execution Paths of a Model) We define $Paths(U) = \{\Lambda \mid Path(U, \Lambda)\}$ as the set of all execution traces of model U.

Let $\Lambda = (\lambda_0, \lambda_1, \lambda_2, \ldots)$. The following path operations will be useful:

$$\Lambda(i) = \lambda_i \quad \Lambda^i = (\lambda_i, \lambda_{i+1}, \lambda_{i+2}, \ldots) \quad \Lambda = \Lambda(0) : \Lambda^1$$

Note that our execution traces contain information about the local state of machine configurations as well as the events generated by execution steps. Therefore, we need to define a property specification language that takes advantage of this information. We have defined such a language in section 3.1. We now define the semantics of a formula Φ with respect to execution path Λ. We write $\Lambda \models \Phi$ if formula Φ is true of execution path Λ. $\models$ is defined inductively on the structure of Φ:

$$\begin{aligned} \Lambda &\models \mathsf{not}\ \Phi && \Leftrightarrow \neg(\Lambda \models \Phi) \\ \Lambda &\models \Phi_1\ \mathsf{and}\ \Phi_2 && \Leftrightarrow (\Lambda \models \Phi_1) \ \wedge \ (\Lambda \models \Phi_2) \\ \Lambda &\models \Phi_1\ \mathsf{or}\ \Phi_2 && \Leftrightarrow (\Lambda \models \Phi_1) \ \vee \ (\Lambda \models \Phi_2) \\ \Lambda &\models \mathsf{next}\ \Phi && \Leftrightarrow \Lambda^1 \models \Phi \\ \Lambda &\models \Phi_1\ \mathsf{until}\ \Phi_2 && \Leftrightarrow \exists i \geq 0.\, (\Lambda^i \models \Phi_2 \wedge \\ & && \qquad \forall 0 \leq j < i . \Lambda^j \models \Phi_1) \\ \Lambda &\models P && \Leftrightarrow \Lambda(0) \models P \end{aligned}$$

We also introduce the usual abbreviations:

$$\begin{aligned} \mathsf{true} &\equiv \Phi\ \mathsf{or}\ (\mathsf{not}\ \Phi) \\ \mathsf{false} &\equiv \Phi\ \mathsf{and}\ (\mathsf{not}\ \Phi) \\ \Phi_1\ \mathsf{implies}\ \Phi_2 &\equiv (\mathsf{not}\ \Phi_1)\ \mathsf{or}\ \Phi_2 \\ \mathsf{eventually}\ \Phi &\equiv \mathsf{true}\ \mathsf{until}\ \Phi \\ \mathsf{always}\ \Phi &\equiv \mathsf{not}\ (\mathsf{eventually}\ \mathsf{not}\ \Phi) \end{aligned}$$

We have divided the definition into two parts. The first part, presented above, deals with the usual temporal logic modalities and reasons about the truth of properties over time. The second part only deals with individual states, as suggested by the definition of $\Lambda \models P$: P is true of the execution path Λ if and only if P is true of its initial state $\Lambda(0)$. We proceed by defining $\lambda \models P$. We start with boolean predicates:

$$(\langle H, \gamma^* \rangle, \alpha) \models b \Leftrightarrow \langle H, \bot, \rho_0 \rangle, b \Downarrow 1$$

where $\rho_0 = [\overline{o}_i \mapsto \hat{o}_i]$, for all objects $\overline{o}_i$ declared in the model. Note that b is evaluated with an environment containing no current object. This is because the predicate is stated at the top level and the only way to access the value of fields is by using objects names, that is, the names used at the top level object declaration. Thus, we create an environment ρ_0 with such bindings. We can write:

$$\begin{aligned} &(\mathsf{always}\ \mathsf{a1.total} < 12)\ \mathsf{and} \\ &(\mathsf{eventually}\ \mathsf{a2.role} = \mathsf{Manager.receiver}) \end{aligned}$$

Let $\sigma = \langle H, \bot, \rho_0 \rangle$. Current states and transitions can be referred by:

$$\begin{aligned} &(\langle H, \gamma^* \rangle, \alpha) \models \mathsf{state}(r, name(s)) \Leftrightarrow \sigma, r \Downarrow \hat{o} \wedge \\ &\qquad \exists \gamma \in \gamma^*, s'.\, \gamma = \langle \hat{o}, -, \hat{s}, -, - \rangle \ \wedge \ s' \in \hat{s} \ \wedge \ s \preceq s' \\ &(\langle H, \gamma^* \rangle, \alpha) \models \mathsf{trans}(r, \bar{t}) \Leftrightarrow \sigma, r \Downarrow \hat{o} \ \wedge \ \mathsf{trans}(\hat{o}, \bar{t}) \in \alpha \end{aligned}$$

For example, the formula eventually state(a2,Branch) is true if, at some point in time, object a2 reaches state Branch. We now define the predicates that deal with message passing:

$$\begin{aligned} (\Gamma, \alpha) \models c(r_1, r_2, m) \Leftrightarrow \sigma, r_1 \Downarrow o_1 \ \wedge \ \sigma, r_2 \Downarrow o_2 \\ c(o_1, o_2, m(-)) \in \alpha \\ \text{where } c \in \{\mathsf{send}, \mathsf{recv}, \mathsf{msg}\} \end{aligned}$$

Note that the arguments of the message are not used by the definition. This is because the scoping rules do not offer a way of binding a method parameter with the value sent by the message. Parameters are, therefore, inaccessible. We solve this by adding the special optional construct L that allows us to evaluate boolean expression with extra local information such us method parameters and a 'current' object. The definition of send is extended in the following way:

$$\begin{aligned} &(\Gamma, \alpha) \models \mathsf{send}(r_1, r_2, m).\mathsf{and}\{g\} \Leftrightarrow \\ &\quad \sigma, r_1 \Downarrow \hat{o}_1 \ \wedge \ \sigma, r_2 \Downarrow \hat{o}_2 \\ &\quad \mathsf{send}(o_1, o_2, m(v_1, \ldots, v_n)) \in \alpha \\ &\quad \rho = \rho_0[x_i \mapsto v_i][\mathsf{SENDER} \mapsto o_1] \\ &\qquad\qquad [\mathsf{RECEIVER} \mapsto o_2][\mathsf{METHOD} \mapsto m] \\ &\quad \sigma' = \langle H, \hat{o}_1, \rho \rangle \ \wedge \ \sigma', g \Downarrow 1 \\ &\text{where } \ c \in \{\mathsf{send}, \mathsf{recv}, \mathsf{msg}\} \end{aligned}$$

Note that the environment σ' contains an object reference - the sender's - and the mapping of parameters to arguments as well as the system variables containing the

values of the sender, receiver and operation name. Similarly, we extend the definition of state:

$$(\Gamma, \alpha) \models \mathsf{state}(r, name(s)).\mathsf{and}\{g\} \Leftrightarrow \\ \exists \gamma \in \gamma^*, s'.\ (\gamma = \langle \hat{o}, -, \hat{s}, -, - \rangle \wedge \\ s' \in \hat{s} \ \wedge \ s \preceq s') \Rightarrow \langle H, \hat{o}, \rho_0 \rangle, g \Downarrow 1$$

We apply similar extensions to the other predicates.

Definition 9 (Model Checking UML Models) We write $U \models \Phi$ if Φ is true at all valid execution paths of U, that is:

$$(U \models \Phi) \Leftrightarrow \forall \Lambda \in Paths(U).\ U \models \Lambda$$

4 Translators and Visualisation

We have implemented the operational semantics and verification of UML models (as defined in this paper) as a translation into Promela, the specification language used by the Spin [12] Model Checker. Our translation takes as input a model U and a formula Φ, and generates a file made of two parts: the specification of the UML model written in Promela, and the Buchi automaton that implements the Spin LTL formula to be verified. The latter is known as a never clause.

The translation of UML models and properties is closely related. On the one hand, the translation of UML models into Promela must implement the UML operational semantics and provide the infrastructure to facilitate the verification (and translation of) of properties that reason about state machines, including the provision of variables to keep track of UML elements such as states, transitions and the events described in section 2.3. On the other hand, property translation must take into account the generated Promela model since it generates Spin LTL formulas that refer to the new Promela variables.

4.1 Translating UML models into SPIN

Promela models are constructed from three basic types of objects: processes, data objects and message channels. Processes, instantiations of proctype declarations, are used to define behaviour. Given a UML model, the translator generates a proctype declaration per class and instantiates one process per object, including field initialisations. Class fields and operation parameters are implemented as data objects; the transformation declares static and non-static fields as global variables and global arrays of structures, respectively, while method arguments are declared as local variables inside the body of the process type declaration of the owning class. Promela message channels are used to model the exchange of data between processes. We use channels to model the input and output queues of state machines, essential parts in the implementation of triggers and method invocation.

```
/* Communication/Message Passing */
mtype = { A0, A1, A2 };
#define QSIZE 2
#define NUMCHAN 3
chan inQ[NUMCHAN] = [QSIZE] of <Msg>;
chan outQ[NUMCHAN] = [QSIZE] of <Msg>;
/* State machines - model checking */
#define NUMOBJECTS 3
byte current[NUMOBJECTS], transition[NUMOBJECTS];
/* Object declarations */
#define a1 0
#define a2 1
#define manager 2
/* Static fields Class RoleMng */
byte RoleMng_J=1,RoleMng_P=2;
/* Static fields Class Peer */
byte Peer_count;
/* Event variables  */
bool _oCall=0;
byte _oSender,_oReceiver,_oTrigger;
byte _out0,_out1,_out2,_out3;
```

where <Msg> = {mtype,byte,byte,byte,byte,byte,byte}

Fig. 10 Peer/Manager Promela Declarations

Promela models generated by our translation have the following structure:

```
<U2P> ::= <GlobalDec>
          <ClassDec>+
          <CommProcDec>
          <InitProcess>
```

The <GlobalDec> section declares all global variables and constants used for communication, state machine execution, object identification, model checking as well as the list of static fields of all classes. The translator generates a <ClassDec> per class and a special proctype declaration for Comm that implements message traffic between objects. Finally, the code generated for <initProcess> instantiates all the objects (processes) that take part of the execution of the system.

We show in Figure 10 parts of the <GlobalDec> section of the Promela model generated for the example. All global declarations start by defining the structures and constants needed for message passing (section 4.1.1). The NUMOBJECTS constant denotes the number of objects declared in the model, and the current and transition arrays store the values of the current state and last executed transition of each state machine. Each object is assigned a unique id *e.g.* a2 is identified by 1, used to index the global arrays holding object information. Each static field is denoted by a global variable composed of the class and field name *e.g.* Manager_J. Finally, The event variables section declares variables used solely for model checking.

A UML class is translated into a global array declaration of a structure that stores the non-static fields of the class, and a Promela process that implements the behaviour defined by its state machine. We show below the top level declarations generated for class Peer:

```
typedef Peer_Fields {byte peer,manager,share,role;};
Peer_Fields fPeer[2];
/* Class number 1  */
proctype Peer(byte pNum, ocID) { atomic {
   <Class-Body>
}}
```

The global array fPeer holds the values of the fields of class Peer. We have declared two objects of that class and, thus, the array has size 2. The Peer process takes two arguments: pNum and ocID. Argument pNum is used to index global arrays with information common to all objects *e.g.* `current`, `transition` and message channels, while index ocID is used to access the fPeer array (section 4.1.2). The values of these indexes are assigned during object instantiation.

```
init { atomic {
  run Peer(0,0);
  run Peer(1,1);
  fManager[0].peer1=a1;
  fManager[0].peer2=a2;
  fManager[0].share=false;
  run Manager(2,0);
  run Comm(3)
}}
```

The object initialisation code `<InitProcess>` generated for our example is shown above. All declared objects are instantiated by executing the `run` Promela construct on the respective `proctype`. Each instantiation is preceded by the explicit field initialisations specified in the model. For example, `manager` is instantiated with a call to `run Manager(2,0)`, preceded by assignments to fields `peer1` and `peer2` *i.e.* `fManager[0].peer1=a1;`.

4.1.1 Message passing and Communication

Communication between processes is performed using channels. The communication model used by our transformation uses two channels per object, one for incoming messages and another for outgoing messages. Object channels are stored in global arrays `inQ` and `outQ`, of type `chan`, and size `NUMCHAN` (see declarations in Figure 10). All messages have the same structure, <operation, sender, receiver, p0,. . .,pn>, with p1,. . .,pn carrying the values passed as arguments to the method call. The special Promela datatype `mtype` is used to define tokens A0, A1 and A2 which denote operations getRole, sendMsg and endTask, respectively. The maximum number of parameters in our example is four and, thus, messages are of type

```
{mtype, byte, byte, byte, byte, bye, byte}
```

Message passing is implemented by writing into the sending object's output queue while incoming messages (triggers) are read from the receiving object's incoming queue. For example, the Spin code generated for sending and receiving messages , respectively, is:

```
outQ[pNum]!trigger(sender,receiver,_out1,..._out4);
inQ[pNum]?trigger(sender,receiver,prm0,...,prm3);
```

The traffic of messages between objects is implemented by a separate process, `Comm`. This is an approach similar to the one used in [13]. The current implementation of `Comm` defines the process as an infinite loop that, at each iteration, removes a message from the output queue of one of the objects of the model and places it, untouched, in the input queue of the matching receiver. If all output queues are empty, the communication process blocks. If more than one input queue has a pending message, `Comm` picks one of them non-deterministically.

The implementation of the `Comm` process for our example is:

```
proctype Comm(byte pNum) {
byte trigger,receiver,sender, prm0,prm1;
end3: atomic {
if
:: outQ[0]?trigger(sender,receiver,prm0,...,prm3) ->
   inQ[receiver]!trigger(sender,prm0,...,prm3);
// repeat for outQ[1] and outQ[2]
fi;
goto end3;
}}
```

By following this approach we provide an alternative - at the implemenation level - to the attack models suggested in Section 3.2. Several types of attacks to the communication channels can be modelled e.g. messages can be dropped, modified or replicated; by manipulating and creating different version of the code that implements `Comm`, as shown with the Dolev-Yao attacker implemented in [15].

4.1.2 Promela implementation of UML classes

A UML class is translated into a Promela process that implements the behaviour defined by its state machine, and updates the variables used for model checking. In this section we describe the implementation of UML classes in SPIN by showing the main parts of the code generated for the Peer class, listed in Figure 11.

On initialisation, a new process (object) receives two values as arguments. The first argument, `pNum`, is used to access global data structures that store information common to all objects such as channels and state machine variables *e.g.* `current[pNum]` contains the value of the current state. The second argument, `ocID`, indexes the global array of structures containing the non-static fields of the class *e.g.* `fPeer[ocID].role` denotes the value of field `role` of the object.

Local declarations - top part of the generated `proctype` - include local variables used to represent formal parameters *e.g.* variables `r` and `p` from method getRole, and variables used for message passing. An object (process) starts execution by setting the `current` variable to the value of the state machine's initial state, and

```
proctype Peer(byte pNum, ocID) { atomic {                        /* Class Peer */
byte txt,m,r,p,s;                      /* Formal Parameters - local variables */
bool msgUnread=false;
byte trigger,receiver,sender,prm0,prm1,prm2;               /* Message passing */
current[pNum] = 0;  transition[pNum]=0;

printf("<t start c=Peer o=%d s=%d >\n",pNum,current[pNum]);          /* Trace */
numStarted++; (numStarted>=3);                                 /* <Synchronise> */
} /* end atomic */

LMAIN0:   /*** Main Loop ***/
atomic {
msgUnread=false; _oCall=false;
if                                      /* Part 1: Completion Transitions */
:: (current[pNum] == 2) ->                                     /* Branch state */
  /* Branch->Send,  Branch->Receive */
:: (current[pNum] == 3) ->                                      /* Write state */
  /* send: Write -> End */
:: (current[pNum] == 4) ->                                      /* Photo state */
  if
  :: (fPeer[ocID].share == true)   ->                           /* Photo -> P2 */
    current[pNum]=5; transition[pNum]=5;
  :: (fPeer[ocID].share == false)   ->                          /* Photo- > P3 */
    current[pNum]=6; transition[pNum]=7;
  fi
:: /* Code for P3 -> Join  Join -> End,  End -> Msg */
:: else -> goto LCOMPLETED0;
fi;
goto LMAIN0;                           /* execute completion transitions first */
LCOMPLETED0:                                          /* check for final state */
if
:: (current[pNum] == 1) -> goto LFINAL0;
:: else -> skip;
fi;
inQ[pNum]?trigger(sender,receiver,prm0,prm1,prm2,prm3);      / * Part 2: Read */
if                                                          /* Parameter passing */
:: (trigger == A0) -> { r = prm0; s = prm1; p = prm2; m = prm3 }
:: (trigger == A1) -> { txt = prm0; }
}
:: else -> skip;
fi;

if                                          /* Part 3: Triggered Transtions */
:: (current[pNum] == 0) ->                                     /* Initial state */
  if
  :: (trigger == A0) ->
    fPeer[ocId].manager = m; fPeer[ocId].peer = p;
    fPeer[ocID].share = s; fPeer[ocID].role = r;
    Peer_count = Peer_count + 1;
    current[pNum]=2; transition[pNum]=2;
  :: else -> msgUnread=true;   transition[pNum]=1;
  fi
:: (current[pNum] == 5) ->                                        /* P1 state */
  if
  :: (trigger == A1) ->
    current[pNum]=6; transition[pNum]=6;
  :: else -> msgUnread=true;   transition[pNum]=1;
  fi
:: (current[pNum] == 9) ->
/* Code for transition Msg -> Pf */
fi;

goto LMAIN0;
LFINAL0: skip;
} }
```

Fig. 11 Promela implementation of class Peer

blocks until the remaining objects have finished initialization. The `printf` statement generates trace information used by the result visualisation module described in section 4.3; we have removed all trace statements from the code shown in Figure 11.

The semantics of state machines is implemented by a loop (`LMAIN0`) that executes until the final state is reached. The main loop is made of three parts: (1) completion transitions, (2) read and (3) triggered transitions. Transition execution, effects included, must be performed atomically. This is achieved by inserting the special Promela atomic statement around the loops that implement the state machine. This is of particular importance because it defines the places where verification takes place; model checking is performed (never automata) after the execution of every atomic statement. Atomicity is broken when a process blocks (message waiting) or when the code jumps out of the scope of the atomic region. We use this fact and introduce back jumps to LMAIN0 outside the atomic area in order to make sure that checks are performed by the verifier after the execution of each transition.

The first part of the main loop implements the execution of transitions that do not contain triggers i.e. completion transitions. This is itself an inner loop that executes until no completion transition is found. Nondeterministic choice is applied if more then one transition is available. The translator maps states, transition and operation names to constants. For example, the branches corresponding to the `Branch` and `Write` states refer to states 1 and 2, respectively. Transition execution updates the values of `current` and `transition`.

The inner loop ends with a check against the final state (state 4 in our model). If the current state is not final, the process reads on its input channel (part 2) and blocks if the queue is empty. When a message arrives, the process unblocks and assigns the values of the message's arguments to the formal parameters variables. The last part of the loop, a conditional that branches depending on the value of the current state and trigger, implements the execution of triggered transitions. If the incoming message does not match any of the available triggers, the message is dropped and the value of `msgUnread` is set to `true`.

Translation of actions is almost straightforward with the exception of the call statement. For example, the code that implements the execution of the action associated to the transition that joins states End and Msg is:

```
/* call peer.sendMsg(''B'');  */
_oTrigger = A1;
_oSender = pNum; _oReceiver = fPeer[ocID].peer;
_oCall=1; _out0=1;   /* ''B'' mapped to constant */
outQ[pNum]!_oTrigger(_oSender,_oReceiver,...,_out3);
current[pNum]=4; transition[pNum]=8;
```

which requires assignments to the message parameters and model checking variables, and a `write` to the object's output channel.

If the model contains instances of the tm guard, the translator does the following:

- Marks all source states of transitions that contain tm.
- Generates declarations for a single global clock variable (`tmGlobal`) and one local timer variable (`tmLocal`) for each class with marked states.
- Generates code that:
  - Sets the local (class) timer to the global clock value whenever a transition reaching a marked state is executed.
  - Increments the global clock after each loop iteration (inside a class process).
  - Increments the global clock if the program blocks. This is done by checking for `timeout` inside the Comm process:

```
:: timeout ->  // added to Comm
  if
  :: (numFinished>=NUMOBJECTS) ->
    goto LCOMMEND
  :: else ->
    tmGlobal++;
  fi
fi;
```

4.2 Translating properties into Spin LTL

Properties written in the property language defined in Section 3.4 must be translated into the LTL version used by Spin. In order to do this, the property translation phase must use the symbol tables used by the model translation phase and refer to the global variables and arrays declared in the generated Promela model. Let's consider the following property:

always (state(a1,Branch) implies
(always a1.role=Manager.J))

The translator generates the following code:

```
#define pp0 (current[0]==2)
#define pp1 (fPeer[0].role==Manager_J)
!([]((pp0) -> ([](pp1))))  // Spin LTL
```

Spin LTL formulas can only deal with boolean variables e.g. predicates like `(x > 2)` are not valid. Therefore, the transformation has to generate special `#define` declarations that name all boolean expressions and plug the new definitions inside the generated Spin formula. In the example above, `pp0` and `pp1` encode the state predicate and the field comparison, respectively. Note that operators are translated into their LTL counterparts e.g. `[]`, `<>`, `U`. The transformed formulas are written using the variables and values used by the generated Promela model e.g. state Branch is denoted by 2, local fields are accesed using the fPeer array, static variables are referenced by the global variable declared in the Promela model, and the global array current is used to check the value of the current state.

Similarly, the translator takes as input the property

```
always send(manager,a2,getRole).implies
                              { role = Manager.P }
```

and generates:

```
#define pp0 ((_oCall==1) && (_oSender==2) &&
             (_oReceiver==1) && (_oTrigger==A0))
#define pp1 (_out0 == Manager_P)
!([]((pp0) -> (pp1)))
```

The generated formula is then transformed into a Büchi automata (never clause) using one of SPIN utilities and both, definitions and never clause, are appended to the model to form the never file.

4.3 Results Visualisation

SPIN provides its imulation and model checking results as text, and in reference to the PROMELA-level specification. This form is unsuitable for software developers as it makes it difficult for them to identify the parts of models involved in the reasoning path that demonstrates property violations. To address this limitation, the translated Promela model contains a series of `printf` statements that generate a trace used later by the SVF. The output of these statements is mixed with the usual Spin messages. The following steps are needed:

- Parse the Spin output and identify the SVF trace messages.
- Parse the messages and translate any Spin specific representation to the corresponding UML model element. For example, state 2 must be transformed into R1.
- Send the transformed output to the user.

The partial output of an execution trace of our sample model is shown in Figure 12. The lines on the right column, with text marked `<t message>`, are generated by the printf instructions added during the code generation phase *e.g.* the `printf` line in Figure 11. These lines are extracted from the output text and all references (states, transitions) are resolved in order to get a final output, shown on the left column, using the following syntax:

- `START o:<Class> state=S` indicates that execution of object o has started in state S.
- `TRANS o:<Class> S1->S2` indicates that the transition that goes from state S1 to state S2 in object o has been executed.
- `OUT: o1->o2 m(<args>)` indicates that message `m(<args>)` has been sent from object o1 to object o2.
- `IN: o2<-o1 m(<args>)` indicates that message `m(<args>)` sent by object o1 has been received by object o2

4.4 Extension to models with composite states

Sections 4.1 and 4.2 describe the translations from UML models and properties into Promela processes and never claims that implement the semantics of the execution and model checking of flat state machines *i.e.* machines with single states. In order to model check state machines that contain composite states the USVF must do the following:

- Applies a flattening algorithm to the initial state machine.
- Generates extra Promela data structures to implement the state predicate.

The flattening algorithm takes as input state machine $\mu = (s_0, s_f, S, T)$ and generates a flat state machine $(s_0, s_f, \mathit{fStates}(\mu), \mathit{fTrans}(\mu))$, where $\mathit{fStates}(\mu)$ is the set of all possible state trees in μ and $\mathit{fTrans}(\mu)$ is the set of new transitions generated from T by taking into considerations the new set of flat states.

The flattening algorithm generates a set of states $\mathit{fStates}(\mu)$ by traversing the state tree and collecting the leave (single) states from simple composite states and performing set product on the flattened states of parallel state machines. The flattening of states is described below:

$$
\begin{aligned}
\mathit{fStates} : \mathrm{SM} &\rightarrow \widehat{\mathrm{State}} \\
\mathit{fStates}(\mu) &= \mathit{fStates}(\mathit{root}) \\
\mathit{fStates}(s) &= \{s\}, s \in \mathrm{Simple} \\
\mathit{fStates}([\overline{s}, R]) &= \mathit{fStates}(R) \\
\mathit{fStates}([\overline{s}, R^*]) &= \textstyle\prod_{R_i \in R^*} R_i \\
\mathit{fStates}(R) &= \{\mathit{fStates}(s) \mid s \in S, R = (\overline{rg}, S)\}
\end{aligned}
$$

The new set of transitions $\mathit{fTrans}(\mu)$ is calculated such that:

$$
\begin{aligned}
(\bar{t}, \hat{s}, \hat{s}', tr, g, a) \in \mathit{fTrans}(\mu) \Leftrightarrow\ & \hat{s}, \hat{s}' \in \mathit{fStates}(\mu) \\
& (\bar{t}, s_I, s_F, tr, g, a) \in T \\
& s'_I \preceq s_I \ \wedge\ s'_I \in \hat{s} \\
& \hat{s} \overset{t}{\rightsquigarrow} \hat{s}'
\end{aligned}
$$

For every transition t in μ with source state s_I we generate a new transition t' for each state $\hat{s}$ such that $s'_I \in \hat{s}$ and $s'_I \preceq s_I$. In other words, the flattening algorithm "copies" t to all flattened substates of S_I.

The presence of composite states requires the gneration of an extra structure in order to implement the state predicate for states other then simple (leaf) states. The verification of $\mathsf{state}(r, \overline{s})$ - where $s \in$ Simple - is implemented by comparing the current state number with the id of s. However, if $\overline{s}$ is a composite state, $\mathsf{state}(r, \overline{s})$ should evaluate to true if the current state number matches any of s's substates id's. We implement this case by generating a boolean array per occurrence of a $\mathsf{state}(r, s)$ with the following specifications:

- The array must be global and of size equal to the total number of flat states in order to be indexed by state id's.

```
       Trace Result                        SPIN output
--------------------------------      --------------------------------
START a1:Peer state=P0                <t start c=Peer o=0 s=0 >
 START a2:Peer state=P0               <t start c=Peer o=1 s=0 >
   START manager:Manager state=R0     <t start c=Manager o=2 s=0 >
   OUT:manager->a1 getRole(1,1)       <t send  s=2 r=0 m=getRole(1,1) >
   TRANS manager:Manager R0->R1       <t trans c=Manager o=2 s=1 >
   TRANS manager:Manager R1->R2       <t trans c=Manager o=2 s=3 >
   OUT:manager->a2 getRole(2,0)       <t send  s=2 r=1 m=getRole(2,0) >
   TRANS manager:Manager R2->Rf       <t trans c=Manager o=2 s=2 >
IN:a1<-manager getRole(1,1)           <t recv  s=2 r=0 m=getRole(1,1) >
TRANS a1:Peer P0->Branch              <t trans c=Peer o=0 s=1 >
 IN:a2<-manager getRole(2,0)          <t recv  s=2 r=1 m=getRole(2,0) >
 TRANS a2:Peer P0->Branch             <t trans c=Peer o=1 s=1 >
TRANS a1:Peer Branch->Photo           <t trans c=Peer o=0 s=2 >
 TRANS a2:Peer Branch->Write          <t trans c=Peer o=1 s=3 >
```

Fig. 12 Partil trace display and SPIN ouput

- All elements of the array are set to false, except those elements corresponding to states $\hat{s}$ such that there exists $s' \in \hat{s}$ and $s' \preceq s$.
- $\mathsf{state}(r, \overline{s})$ is implemented by accesing the array with the current state number.

The boolean arrays are generated for every predicate and inserted to the Promela code together with the never claim.

5 Implementation

5.1 Implementation

The USVF is packaged as an Eclipse plug-in that runs along the Papyrus UML [31] graphical modeler. The operational semantics UML models is implemented as a translation into Promela, the specification language used by the Spin [12] Model Checker. The translation of both models and properties is integrated with a UML graphical editor (Papyrus), a property editor and a result visualisation component that interact with Spin and the user [37]. Model simulation and verification is performed by Spin.

The USVF and its graphical user interface are implemented in Java and SWT, the GUI toolkit used by the Eclipse platform. All parsers were generated using JavaCC.

The Design Model Constructor is responsible for the creation of UML models, the generation of the internal model representation and basic model validation. Papyrus UML is the Graphical UML editor chosen to run along the USVF. Papyrus has the required functionality that will allow the user to build UML class and state machine diagrams, including provision for the definition of transition guards, effects and actions.

The core functionality for model construction is provided by the Papyrus UML plug-in. The main output of this component is the XMI representation of the model, saved as a `.uml` file, which is later read and parsed. Internal representation generation is divided into three steps: (1) Parsing of the XML model file generated by Papyrus (.uml) and parsing of the guard and effects in transitions. The latter requires a separate parser, (2) Construction of internal representation of UML model with most references between objects still missing, (3) Resolution of all internal references and generation of complete internal representation, and (4) Soundness check e.g. minimal number of classes, states, etc.

In essence, internal representation generation performs type checking on the original model. If successful, the translated model shall execute without runtime type errors.

The USVF performs verification of Papyrus UML models against properties specified in the property specification language defined in section 3.4. The USVF gets this input from the user, parses it and translates it to its internal representation. An important part of this process is the resolution of formula elements to elements in the UML model internal representation, effectively type checking the property formula against the loaded model.

The implementation of the model and property translators is described in section 4. The outputs of both translators are put together into a single file, the never file, which is used as input by the model checker.

The USVF uses Spin to model check system and security properties against the UML models generated with Papyrus UML. Spin models are written in Promela, a special language similar to C. Spin can perform simulations directly on Promela files. Verification is a more complicated process and takes three steps. Given a Promela file, Spin generates a C file with the code that implements the verifier for that particular model. The C program is compiled into a `pan` file. The `pan` file is executed.

The USVF provides an interface to Spin implemented as a wrapper class. This wrapper class makes all the necessary system calls to Spin, the gcc compiler and the compiled verifier. Through the wrapper class, USVF uses the Spin model checker to:

- Simulate Promela models.

– Call the Spin never clause generator which transforms LTL formulas into never clauses.
– Generate a verifier executable given a Promela model containing the property to be verified (never clause).
– Execute the verifier.
– Execute a simulation on the trail generated by the verifier in case the verification fails.

6 Related Work

6.1 Secure Software Engineering and Modelling Languages

The area of secure software engineering has produced solutions that aim to help system designers address security issues during the whole development life-cycle of software systems. In particular, UMLsec [14,16] and SecureUML [22] tackle this problem by introducing security requirements and constraints in the design phase via annotations on UML models.

UMLsec and SecureUML are based on UML - profiles *i.e.* , a set of stereotypes, tagged values and constraints - for modelling security properties. Each of these approaches focuses on specific types of properties. More specifically, SecureUML focuses on role-based access control (RBAC) and supports the specification of authorisation constraints. It combines the simplicity of using UML's graphical notation as the basis for expressing RBAC, with the power of dynamic authorisation constraints, *i.e.* , contraints based on the state of the system e.g. field and parameter values. In [22], Lodderstedt *et.al.* show how SecureUML specifications can be used to generate security infrastructures that implement RBAC.

UMLsec provides a series of stereotypes used to model security-related characteristics of system components (communication links, roles, guarded elements) as well as security requirements of systems (secrecy, integrity, information flow, fair exchange, RBAC). In [15,16], Jürjens *et.al.* show how UMLsec annotations can be used to automantically evaluate UML specifications for vulnerabilities using a formal semantics of a simplified fragment of UML and model checking techniques. In particular, they address privacy by model checking an automatically generated Promela model with cryptography operators and that includes a Dolev-Yao attacker.

Current research efforts in the area of secure software engineering include the integration of security methodologies and specification techniques. For example, in [27] Mouratidis *et.al.* merge the high-level concepts and modelling activities of the secure Tropos methodology with UMLsec models. The approach of USVF - unlike UMLsec and SecureUML - is not based on the introduction of a special purpose profile. Instead, in USVF we introduce a generic property language that can be used to express not only basic security properties but also more generic liveness and safety property as we explained in Section 3.1. In principle, methodologies like UMLsec and SecureUML can be integrated to USVF by using the property specification language of the latter framework as an intermediate language between the security properties specified by the aforementioned profiles and the model checker. Furthermore, USVF complements UMLsec and SecureUML by enabling the specification of a wide range of verification properties (and their association with UML model elements) that can not be inferred or generated from the UML tags and stereotypes used in UMLSec and SecureUML. For example, properties such as the availability of a paritcular service (liveness) under a particular set of conditions, or the constraints on *e.g.* the values of fields, at specific points during the execution of a model need to be specified by the designer using a language such as the USVF property specification language.

6.2 UML Semantics and Model Checking

The need to develop a more precise specification of UML has been a concern [7] since its inception and adoption as standard notation for object-oriented analysis and design by the Object Management Group (OMG). As a result, several formalisations have been proposed for the behavioural part of UML and, in particular, the formal specification of the semantics of state machine diagrams [29,3,13].

Most of the work on formalisation of UML state machines has been in the context of automated formal verification of systems and, in particular, model checking [17, 30,35,20]. A good number of these specifications have been used as input to translations into Spin.

The automatic verification of UMLsec models, described in [16,15], is the most thorough work on model checking security requirements of UML models using Spin. The Spin translation used in this paper (for flat state machines) resembles the one defined in [16]. Both, USVF and UMLsec, consider non-hierarchical state machines, treat completion and triggered transitions separately (in the main loop), and define a separate process for message exchange. The latter is used to model intruders in conjunction with a cryptographic action language. In UMLsec, model checking is instructed by a series of annotations based on stereotypes, which are transformed directly into SPIN LTL. Our main contribution with respect to the UMLsec translation is that our framework allows the user to explicitly write properties, using the property specification language, and that the transformation generates special code to model check the predicates specified by the specification language *e.g.* special SPIN variables are defined for keeping track of field updates, sending and reading of messages.

Jussila et.al. [17], like us, model UML classes as SPIN processes and define a separate action language. However, they do not provide a UML-based specification

language; the user is limited to entering checks in SPIN LTL. The Hugo project [35] on the other hand uses a Spin translation to verify collaboration diagrams against UML state machines but they do not support the verification of user-defined temporal properties against the model as in USVF.

Some of the work related to formal specification of semantics of UML has been dedicated to provide a complete formalisation of complex aspects of state machines such as hierarchical state machines and history states. For example, Latella *et.al.* [21] model UML state machine diagrams as extended hierarchical automata using Kripke structures. Our goal has been to formalise a simple, though expressive, subset of UML in such a style that faciliates the definition of UML model checking *i.e.* small step semantics with labeled transitions. Along those lines, our UML semantic specification resembles the work presented in [3]. In particular, starting from a precise textual syntax definition, they develop a concise structured operational semantics for UML-Statecharts based on labeled transition systems. Our approach to the modelling of composite states follows the lines of the work of Gnesi *et.al.* [8] and Kuske [19] who use the concepts of trees and term-rewriting. We simplify the latter by representing state trees with the set of the tree's simple substates (leaves).

In [40], Xie *et.al.* transform models expressed in xUML, an executable subset of UML, into S/R models that can be verified by the COSPAN model checker. COSPAN is an ω-automata-based model checker that takes as input models and queries formulated in S/R (in their work, models are defined as synchronous parallel composition of processes). One of the most attractive points mentioned is the use of static partial order reduction for model optimisation. However, no details of the approach are included. Similarly, no syntax for the property specification language is included which, from the examples given, seems to be limited to conditions about machine states.

In [24], Moller *et.al.* describe how CSP-OZ, a formal method combining the process algebra CSP with the specification language Object-Z, can be integrated into an object-oriented software engineering process employing UML as a modelling language and Java as an implementation language. Their methodology considers the use of runtime checking tools to supervise the adherence of the final Java implementation to generate JML contracts. However, unlike in USVF, the types of properties considered are not temporal. More specifically, the static verification part of their approach deals with JML-style annotations while the dynamic verification part checks local assertions and invariants.

The most advanced work in UML model checking is the one developed by Gnesi *et.al.* [9,4]. In [9], they define a logic based on μ-ACTL, a state/event-based temporal logic similar to the one in USVF that uses a doubly labeled transition system as semantics domain. They also implement an on-the-fly model checker and report application of the framework to different application domains such as verification of protocols for service-oriented systems [4]. However, the property language of USVF is able to express a richer set of predicates involving object fields, class fields and action events, aided by the operational semantics exposure of more action-related events and richer syntax.

7 Conclusions and Future Work

In this paper we have presented USVF, a framework that allows software developers to build and verify UML models against properties specified in a general-purpose property language. We propose the specification and verification of security and general system properties, as well as the use of formal verification techniques, from the early stages of software development.

We have defined the syntax and semantics of a property specification language for UML model checking. In order to do this we have defined the operational semantics of UML models, provided a property specification language based on LTL and UML elements, and expressed the semantics of UML properties in terms of the runtime domains used by the operational semantics. This approach allows us get into the details of UML model execution thus increasing the type of properties to be verified.

USVF was evaluated in terms of usability, performance, and expressiveness of the property language by the industrial partners in the PEPERS [32] project. The results of this evaluation were positive. The partners were particularly satisfied with expressiveness of the specification language and the integration of the model checking phase with the model creation tool (Papyrus). They also highlighted the fact that in order to have a complete framework to support development and analysis of security aspects of the system, it is necessary to include a way of handling cryptographic primitives. This can be achieved by extending the action language of USVF.

The achievements, and limitations, presented in this paper set up the basis for interesting future work. As mentioned in section 2.1, the UML subset used by the USVF includes the basic features necessary to model communicating state machines. However, in order to improve the usability of USVF, the UML models considered in this paper should be extended to include features such as synchronous operation invocation, as well as concepts related to distributed computing *e.g.* ports. Furthermore, USVF does not have, besides object declarations, a pre-determined way of initialisating objects. The use of constructors and component diagrams - that include, for example, constraints in the number of objects and initialisation conditions - would be an important addition to the framework.

The area of UML model checking offers interesting topics of further research. In particular, we would like the investigate other approaches to the translation of hierarchical state machines to Promela, besides flattening, and their effect in model checking. We would also like to explore the possibility of the application of program analysis techniques, such as slicing [11], for the generation of more efficient models *i.e.* models optimised for model checking of formulas given as input.

Further work also includes the addition inclusion of UML stereotypes for the specification of security requirements and properties for domain specific applications such as service-oriented systems.

Finally, the formal verification of design models should be complemented with further verification steps in order to cover the complete software development process [24]. These steps include the static verification of implementations (where the models become specifications) and the use of dynamic verification techniques [38,18] such as non-intrusive runtime monitoring. In particular, we would like to explore the relationship with latter with the static verification of UML models.

References

1. Abadi, M., Blanchet, B., Fournet, C.: Just fast keying in the pi calculus. In: 13th European Symposium on Programming (ESOP04, pp. 340–354. Springer (2004)
2. Anderson, R.J.: Security Engineering: A Guide to Building Dependable Distributed Systems. Wiley Publishing (2008)
3. von der Beeck, M.: A structured operational semantics for uml-statecharts. Software and System Modeling **1**(2), 130–141 (2002)
4. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: An action/state-based model-checking approach for the analysis of communication protocols for service-oriented applications. In: FMICS, pp. 133–148 (2007)
5. Devanbu, P.T.: Software engineering for security: a roadmap. In: The future of Software Engineering, pp. 227–239. ACM Press (2000)
6. Emerson, E.: Temporal and modal logic. In: J.V. Leeuwen (ed.) Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. MIT Press (1990)
7. Evans, A., Bruel, J.M., France, R., Lano, K., Rumpe, B.: Making UML precise. In: L. Andrade, A. Moreira, A. Deshpande, S. Kent (eds.) Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How? (1998). URL citeseer.ist.psu.edu/evans98making.html
8. Gnesi, S., Latella, D., Massink, M.: Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. Journal of Logic and Algebraic Programming **51**(1), 43–75 (2002)
9. Gnesi, S., Mazzanti, F.: On the fly model checking of communicating UML state machines. In: ACIS. IEEE (2004)
10. Gritzalis, S., Spinellis, D., Georgiadis, P.: Security protocols over open networks and distributed systems: Formal methods for their analysis, design, and verification. Computer Communications **22**, 70–7 (1999)
11. Hatcliff, J., Dwyer, M., Zheng, H.: Slicing software for model construction. Higher-order and symbolic computation **13**(4), 315–353 (2000)
12. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
13. Jürjens, J.: A UML statecharts semantics with message-passing. In: Applied Computing 2002, pp. 1009–1013. Madrid (2002). Proceedings of the 2002 ACM Symposium of Applied Computing
14. Jürjens, J.: Secure Systems Development with UML. Springer-Verlag, Berlin Heidelberg New York (2004)
15. Jürjens, J., Shabalin, P.: Automated verification of UMLsec models for security requirements. In: T. Baar, A. Strohmeier, A. Moreira, S.J. Mellor (eds.) UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings, *LNCS*, vol. 3273, pp. 365–379. Springer (2004)
16. Jürjens, J., Shabalin, P.: Tools for secure systems development with UML. International Journal on Software Tools for Technology Transfer **9**(5), 527–544 (2007)
17. Jussila, T., Dubrovin, J., Junttila, T., Latvala, T., Porres, I.: Model Checking Dynamic and Hierarchical UML State Machines. In: D. Hearnden, J.G. S, B. Baudry, N. Rapin (eds.) MoDeVa: Model Development, Validation and Verification. University of Queensland, Le Commissariat l'Energie Atomique - CEA (2006)
18. Kloukinas, C., Spanoudakis, G.: A pattern-driven framework for monitoring security and dependability. In: TrustBus, pp. 210–218 (2007)
19. Kuske, S.: A formal semantics of UML state machines based on structured graph transformation. In: UML 2001: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, pp. 241–256. Springer-Verlag, London, UK (2001)
20. Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. Formal Asp. Comput. **11**(6), 637–664 (1999)
21. Latella, D., Majzik, I., Massink, M.: Towards a formal operational semantics of uml statechart diagrams. In: Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), p. 465. Kluwer, B.V., Deventer, The Netherlands, The Netherlands (1999)
22. Lodderstedt, T., Basin, D.A., Doser, J.: Secureuml: A uml-based modeling language for model-driven security. In: UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language, pp. 426–441. Springer-Verlag, London, UK (2002)
23. Meadows, C.: Formal verification of cryptographic protocols: A survey. In: ASIACRYPT, pp. 135–150 (1994)
24. Möller, M., Olderog, E.R., Rasch, H., Wehrheim, H.: Integrating a formal method into a software engineering process with UML and Java. Formal Aspects of Computing **20**(2), 161–204 (2008). DOI http://dx.doi.org/10.1007/s00165-007-0042-7
25. Mouratidis, H., Giorgini, P.: Integrating Security and Software Engineering: Advances and Future Vision. IGI Global (2006)
26. Mouratidis, H., Giorgini, P., Manson, G.: When security meets software engineering: a case of modelling secure information systems. Information Systems **30**(8), 609–629 (2005)
27. Mouratidis, H., Jürjens, J., Fox, J.: Towards a comprehensive framework for secure systems development. Advanced Information Systems Engineering pp. 48–62 (2006)
28. Object Management Group. http://www.uml.org
29. Paltor, I., Lilius, J.: Formalising uml state machines for model checking. In: R.B. France, B. Rumpe (eds.) UML 1999, *Lecture Notes in Computer Science*, vol. 1723, pp. 430–445. Springer (1999)

30. Paltor, I.P., Lilius, J.: vUML: A tool for verifying UML models. In: R.J. Hall, E. Tyugu (eds.) Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99. IEEE (1999)
31. Papyrus UML. http://www.papyrusuml.org
32. PEPERS project. http://www.pepers.org
33. Pfleeger, C.P., Pfleeger, S.L.: Security in Computing. Prentice Hall PTR, Upper Saddle River, NJ, USA (2006)
34. R, J.K., Mathur, A.P.: Software engineering for secure software - state of the art: A survey. Tech. rep., Purdue University (2005)
35. Schäfer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. Electronic Notes in Theoretical Computer Science **55**(3), 13 pages (2001)
36. Schneider, F.: Enforceable security policies. ACM Transactions on Information and Systems Security **3(1)** (2000)
37. Siveroni, I., Spanoudakis, G., Zisman, A.: Property specification and static verification of UML models. In: Proc. 3rd International Conference on Availability, Reliability and Security (ARES 2008). IEEE Computer Society, Barcelona (2008)
38. Spanoudakis, G., Kloukinas, C., Androutsopoulos, K.: Towards security monitoring patterns. In: SAC, pp. 1518–1525 (2007)
39. Wynskel, G.: The Formal Semantic of Programming Languages. MIT Press (1993)
40. Xie, F., Levin, V., Browne, J.C.: Model checking for an executable subset of uml. Automated Software Engineering, ASE 2001 p. 333 (2001)